目次

- GitHub 專案連結：https://github.com/telunyang/python_nlp

Module 1. 何謂自然語言處理(Natural Language Processing)

簡介

自然語言處理（Natural Language Processing, NLP）是人工智慧與電腦科學中的一個核心領域，目標是讓電腦能夠「理解、分析、生成」人類語言（例如中文、英文等），並把語言轉換成可計算、可推論的資訊。

NLP 的應用涵蓋日常生活與專業場景：像是搜尋引擎的查詢理解與排序、機器翻譯、語音助理的對話理解、情緒分析、文件分類與摘要、問答系統，以及近年廣泛使用的大型語言模型（LLM）所驅動的文字生成與多輪對話。典型的 NLP 流程通常包含資料蒐集與清理、文字前處理（分詞、斷句、正規化等）、特徵表示（從傳統的詞袋/TF-IDF 到詞向量與深度語言表示）、模型訓練與評估，最後部署到實際產品或研究系統中。

隨著深度學習與 Transformer 架構的成熟，NLP 已從以規則與統計為主，逐步演進到以預訓練模型為核心的典範，顯著提升了跨任務、跨領域的遷移能力，但同時也帶來偏誤、可解釋性、資料隱私與生成內容可靠性等新的挑戰。

文字的特性

在自然語言處理（NLP）中，文字的特性包括以下幾點：

- **不定長度**：
  每個句子或段落的長度可能會有所不同，這使得處理文字比處理固定長度的資料（如圖像）更為複雜。
- **順序性**：
  文字的順序對於其意義具有重要影響。例如，「狗追貓」和「貓追狗」雖然包含相同的詞，但意義卻完全不同。
- **語義多樣性**：
  一個詞在不同的語境中可能具有不同的意義，這就需要 NLP 技術理解和考慮語境。例如，「蘋果」可能指的是水果，也可能是電腦公司；「bank」可能指的是銀行，也可能是河岸邊(河畔)。
- **語言特性**：
  不同的語言具有不同的結構和規則。例如，英語中的字詞是由空格分隔的，而中文則沒有明顯的詞之間的分隔符號。
- **文化和語境因素**：

文字所傳達的意義經常與其文化和語境有關。這使得 NLP 需要理解文化和語境，才能準確地處理和理解文字。

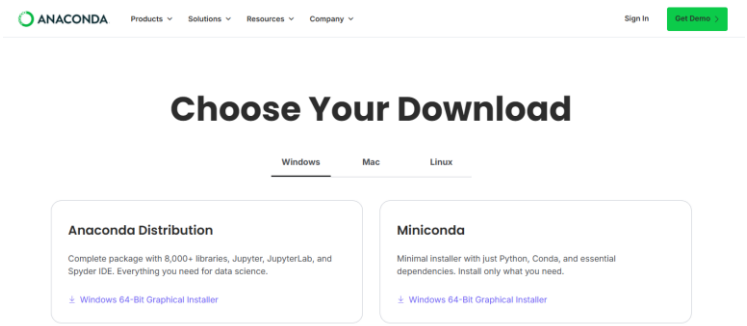
常見用途

自然語言處理（NLP）在許多領域中都有著廣泛的應用，以下列舉幾個常見的例子：

- **機器翻譯**：
  例如 Google 翻譯，能自動將一種語言的文字翻譯成另一種語言。
- **情感分析**：
  用來判定文字內容中隱含的情感傾向，例如正面、負面或中性。這在社交媒體互動、品牌管理等領域都有廣泛的應用。
- **聊天機器人**：
  需要理解用戶所輸入的語言文字，並生成適當的文字回應。這在客戶服務、智慧型助手等領域都有廣泛的應用。
- **文本生成**：
  可以用來自動生成文章、報告、詩歌等。例如，新聞公司可以自動撰寫財經報告。
- **文本摘要**：
  將長篇文章自動壓縮成短小的摘要，以方便使用者快速獲取所需資訊。
- **問答系統**：
  像是 ChatGPT、Gemini、Claude、Grok 等大型語言模型，利用 NLP 來理解用戶的問題並提供相對應的答案。

下載 miniconda

頁面：https://www.anaconda.com/download/success



圖：點下 Miniconda 的 Windows 64-Bit Graphical Installer 連結

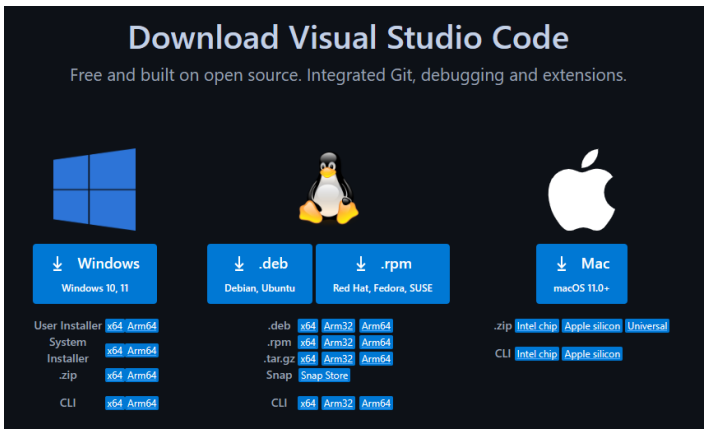
安裝 conda 的 nlp 虛擬環境
conda create --name nlp python=3.12 ipykernel

Optional: 刪除 nlp 虛擬環境
conda remove --name nlp --all

安裝套件
pip install -r requirements.txt

下載 Visual Studio Code

網址：https://code.visualstudio.com/Download



圖：根據不同作業系統來選擇，在這裡我們選擇 Windows 10,11 連結

Module 2. 文字預處理

流程說明

不同的語言在自然語言處理的預處理階段會有些不同；這只是預處理的基本步驟，根據不同的應用和資料集，可能需要調整或增加更多的步驟：

英文 NLP 預處理

- **文本清洗**：
  去除文本中的無關內容，如 HTML 標籤、特殊字符等。
- **分詞**（Tokenization）：
  將句子分割為字詞或詞語。

- **轉換為小寫**：
  將所有字詞轉換為小寫，這有助於統一詞彙。
- **去除停用詞**：
  去除一些在語句中出現頻率高，但對於語義分析沒有多大貢獻的詞，如“is”，“and”，“the”等。
- **詞幹提取(Stemming) & 詞性還原(Lemmatization)**：
  將字詞轉換為其基本形式。例如，詞幹提取可能將"eating"、"eats"轉換為"eat"；詞性還原可能將"running"、"ran"在指定詞性(例如動詞verb)後，轉換為"run"。
- **詞袋(bag-of-word, BoW)**或**詞嵌入(Word Embedding)**：
  將文字轉換為模型能理解的數值或向量。

中文 NLP 預處理

- **文本清洗**：
  同樣需要去除文本中的無關內容。
- **分詞**：
  中文的分詞較為複雜，因為詞與詞之間沒有明顯的分隔符。常見的中文分詞工具包括 jieba 等。
- **去除停用詞**：
  與英文相似，中文也有一些常出現但意義較小的詞需要去除，例如「你、我、他、的」這些經常出現在文章段落中出現的字詞。
- 轉換為**詞向量(Word Vector)**：
  這可以使用詞袋模型(Bag-of-word, BoW)，也可以使用詞嵌入(Word Embedding)技術如 Word2Vec。

英文字預處理

NLTK（Natural Language Toolkit）是一個 Python 函式庫，用於處理自然語言處理（NLP）的相關任務。它提供了各種工具和資源，包括語料庫、語法分析器、詞性標註器、詞幹提取器等，可用於文本處理和分析。

2-1 牛刀小試

```
# 匯入套件
import nltk
from nltk.corpus import stopwords, wordnet
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
```

```
from nltk import pos_tag, ne_chunk
import re

# 下載分詞器
nltk.download("punkt_tab")

# 下載停用詞
nltk.download('stopwords')

# 下載已經定義字詞
# 語義關係來尋找上下位關係的英文詞典，
# 也包含了同義詞、時態、名詞單複數等資訊
nltk.download('wordnet')

# 下載詞彙關係資料庫
nltk.download("omw-1.4")

# 下載詞性標註器
nltk.download("averaged_perceptron_tagger_eng")

# 下載命名實體辨識所需資源
nltk.download('maxent_ne_chunker_tab')

# 下載詞彙表
nltk.download('words')

# 輸入文本
text = '''\
When I was young
I'd listen to the radio
Waitin' for my favorite songs
When they played I'd sing along
It made me smile.
Those were such happy times
And not so long ago
How I wondered where they'd gone
But they're back again
```

```
Just like a long lost friend
All the songs I loved so well.
Every Sha-la-la-la
Every Wo-o-wo-o
Still shines
Every shing-a-ling-a-ling
That they're startin' to sing's
So fine.
'''

# 分詞
# preserve_line=True 保留換行符號
tokens = word_tokenize(text, preserve_line=True)

# 轉換為小寫
tokens = [word.lower() for word in tokens]

# 只保留英文單字
tokens = [t for t in tokens if re.fullmatch(r"[a-z]+", t)]

# 去除停用詞
stop_words = set(stopwords.words('english'))
tokens = [word for word in tokens if word not in stop_words]

# 取得詞性標註
tagged = pos_tag(tokens)

# 命名實體辨識 (Named Entity Recognition, NER)
'''
補充：
如果你想進行 NER，也可以考慮使用更強大的工具，例如 SpaCy、Stanza 或
transformers（Hugging Face）。
'''
ne_chunks = ne_chunk(tagged)

# 詞形還原
lemmatizer = WordNetLemmatizer()
```

```
# 預設名詞
print(lemmatizer.lemmatize(word="played"))

# 指定動詞
print(lemmatizer.lemmatize(word="played", pos=wordnet.VERB))

'''
什麼時候需要用到下面這段程式碼？
- 建議使用（傳統 NLP / 特徵工程場景）
- 你要做詞頻、TF-IDF、BM25、傳統分類/分群等
- 你想降低 vocabulary（把 play/played/playing 合併）
- 你在做 IR（資訊檢索）想提高匹配率
'''

# 將 Penn Treebank 標註轉換為 WordNet 可接受的標註
def penn_to_wordnet(tag: str):
    if tag.startswith("J"): # JJ, JJR, JJS ... (形容詞，副詞比較級，最高級)
        return wordnet.ADJ
    if tag.startswith("V"): # VB, VBD, VBG, VBN, VBP, VBZ ... (動詞過去式，現在分詞，過去分詞)
        return wordnet.VERB
    if tag.startswith("N"): # NN, NNS, NNP, NNPS ... (名詞單數、複數)
        return wordnet.NOUN
    if tag.startswith("R"): # RB, RBR, RBS ... (副詞，副詞比較級，最高級)
        return wordnet.ADV
    return wordnet.NOUN  # fallback

# 詞形還原
tokens_lemm = [lemmatizer.lemmatize(word=w, pos=penn_to_wordnet(t))
for w, t in tagged]

# 顯示前 50 個詞形還原結果
tokens_lemm[:50]
```

補充：詞性對照表

Tag	Description	Tag	Description
CC	conjunction, coordinating	PRP$	pronoun, possessive
CD	numeral, cardinal	RB	adverb
DT	determiner	RBR	adverb, comparative
EX	existential there	RBS	adverb, superlative
FW	foreign word	RP	particle
IN	preposition, conjunction	SYM	symbol
JJ	adjective or numeral, ordinal	TO	'to' prep. or infin. marker
JJR	adjective, comparative	UH	interjection
JJS	adjective, superlative	VB	verb, base form
LS	List item marker	VBD	verb, past tense
MD	modal auxiliary	VBG	verb, pres. participle, gerund
NN	noun, common, sing. or mass	VBN	verb, past participle
NNP	noun, proper, sing.	VBP	verb, pres. tense, not 3rd sing.
NNPS	noun, proper, plural	VBZ	verb, pres. tense, 3rd sing.
NNS	noun, common, plural	WDT	WH-determiner
PDT	pre-determiner	WP	WH-pronoun
POS	genitive marker	WP$	WH-pronoun, possessive
PRP	pronoun, personal	WRB	Wh-adverb

參考資料：

[1] NLTK 初學指南(一)：簡單易上手的自然語言工具箱－探索篇
https://medium.com/pyladies-taiwan/nltk-初學指南-一-簡單易上手的自然語言工具箱-探索篇-2010fd7c7540

[2] Williams, S. (2013). An Analysis of POS Tag Patterns in Ontology Identifiers and Labels.
https://oro.open.ac.uk/90307/1/TR2013-02.pdf

[3] 詞性標註
https://github.com/fxsjy/jieba?tab=readme-ov-file#%E4%BD%BF%E7%94%A8%E7%A4%BA%E4%BE%8B

[4] 真 IT 小叮噹深度學習：自然語言處理（二）文本處理流程
https://kknews.cc/tech/xlmxzmo.html

[5] Extract Entities From Text In Natural Language Processing
https://medium.com/@nutanbhogendrasharma/extract-entities-from-text-in-natural-language-processing-66c77169973d

中文字預處理

2-2 中文字預處理

```
# 匯入套件
import jieba
```

```
# 輸入文本
text = "這是使用 Jieba 和 sklearn 進行中文字預處理的範例。"

# 分詞 (cut 是用於 iteration 的斷詞函式，lcut 是將斷詞以 list 格式回傳)
tokens = jieba.lcut(text)

# 去除停用詞，這邊為了簡化，我們預先定義一個停用詞列表
stop_words = ['和', '的', '是', '這']
tokens = [word for word in tokens if word not in stop_words]

# 讀取自定義的字典
# 格式: 詞語 詞頻(可省略) 詞性(可省略)
jieba.load_userdict('./dict.txt')

# 自定義字詞 (執行一次就會存在 memory 中，要重啟 kernel 才會失效)
jieba.add_word('網球史上')
jieba.add_word('網球運動員')
jieba.add_word('單打冠軍')
jieba.add_word('冠軍')
jieba.add_word('大滿貫')
jieba.add_word('總決賽')
jieba.add_word('大師賽')

# Optional: 你也可以移除自定義的字詞
jieba.del_word('網球史上')

# 輸入文本
text = '''羅傑費德勒，已退役的瑞士男子職業網球運動員，費德勒總共贏得 20 座大滿貫冠軍，單打世界排名第一累計 310 周，其中包括連續 237 周世界排名第一的男子網壇紀錄，為網球史上最佳的男子選手之一。費德勒生涯贏得 103 個 ATP 單打冠軍，含 20 座大滿貫冠軍和 6 座 ATP 年終總決賽冠軍，以及 28 座大師賽冠軍。'''

# 分詞 (cut 是用於回傳 generator 的斷詞函式，lcut 是將斷詞以 list 格式回傳)
tokens = jieba.lcut(text)
```

```python
# 去除停用詞，這邊為了簡化，我們預先定義一個停用詞列表
stop_words = ['和', '的', '是', '這']
tokens = [word for word in tokens if word not in stop_words]

# POS tagging
import jieba.posseg as pseg

# 假設我們有以下句子
sentence = "我喜歡看電影"

# 使用 jieba 進行詞性標注
words = pseg.cut(sentence)

# 列印每個詞及其詞性
for word, flag in words:
    print(f'{word} {flag}')
```

補充：Jieba 的中文詞性對照表

標籤	意義	標籤	意義	標籤	意義	標籤	意義
n	普通名詞	f	方位名詞	s	處所名詞	t	時間
nr	人名	ns	地名	nt	機構名	nw	作品名
nz	其他專名	v	普通動詞	vd	動副詞	vn	名動詞
a	形容詞	ad	副形詞	an	名形詞	d	副詞
m	數量詞	q	量詞	r	代名詞	p	介詞
c	連接詞	u	助詞	xc	其他虛詞	w	標點符號
PER	人名	LOC	地名	ORG	機構名	TIME	時間

參考資料：

[1] jieba

https://github.com/fxsjy/jieba

[2] 中文斷詞

https://blog.maxkit.com.tw/2020/08/blog-post.html

[3] 彙整中文與英文的詞性標註代號：結巴斷詞器與 FastTag / Identify

the Part of Speech in Chinese and English
https://blog.pulipuli.info/2017/11/fasttag-identify-part-of-speech-in.html
[4] POS tagging 词性标注 之 武林外传版
http://codewithzhangyi.com/2019/03/12/pos/
[5] [Day 12] 時間都去哪了？資料前處理：抓住那個欠錢不還的傢伙－詞性標註之 HMM 的應用
https://ithelp.ithome.com.tw/m/articles/10298368

補充：詞雲(Word Cloud)

Word Cloud（又稱文字雲、字雲、詞）是一種視覺化工具，用於表示一組文本資料中的詞彙頻率(很常用在特定文章中，檢視哪些用字遣詞比較常見，藉此了解特定議題的聲量)。在 Word Cloud 中，每個字詞的大小代表它在文本中出現的頻率或重要性。出現頻率較高的字詞會以較大的字體呈現，而出現頻率較低的字詞則以較小的字體呈現。這種方式可以快速地向讀者呈現文本的主要主題。

GitHub: https://github.com/amueller/word_cloud
Doc: https://amueller.github.io/word_cloud/index.html

2-3 詞雲(Word Cloud)

```
'''
基本用法
'''
from wordcloud import WordCloud
import matplotlib.pyplot as plt

# 假設我們已經有了一個字詞和它的頻率的字典
word_freq = {'Python': 50, '資料科學':40, '機器學習': 30, '深度學習': 20, '自然語言處理': 5}

# 建立一個文字雲物件
wc = WordCloud(
    font_path='./fonts/NotoSansTC-Regular.otf',
    background_color='white',
    width=1024,
```

```
    height=768
)

# 生成文字雲 (wc.generate_from_frequencies 用於 dict)
wc.generate_from_frequencies(word_freq)

# 顯示文字雲
plt.figure(figsize=(10.24, 7.68), dpi=150)
plt.imshow(wc, interpolation='bilinear')
plt.axis('off')
plt.show()


'''
使用自訂圖片來作為詞雲顯示範圍
'''
from wordcloud import WordCloud, ImageColorGenerator
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np
import jieba

# 讀取圖片
image = Image.open('./images/doraemon_01.png')  # 替換為你的圖片路徑
mask = np.array(image)

# 這是我們的原始文本資料，這裡只是一個例子，你可以使用任何你想要分析的文本
text = '''
謝 天

常到外國朋友家吃飯。當蠟燭燃起，菜餚布好，客主就位，總是主人家的小男孩或小女孩舉起小手，低頭感謝上天的賜予，並歡迎客人的到來。
…
…
'''

# 使用 jieba 進行分詞
```

```
seg_list = jieba.lcut(text)

# 將分詞後的結果組合成一個長字串，詞與詞之間以空格相隔，這是 wordcloud 需要的格式
seg_str = ' '.join(seg_list)

# 建立一個詞雲物件
wc = WordCloud(
    font_path='./fonts/NotoSansTC-Regular.otf',
    background_color='white',
    mask=mask, # 別忘了設定 mask，否則會出現矩形詞雲，不會是你想要的形狀
    width=820,
    height=1349,
    random_state=42
)

# 生成詞雲 (wc.generate 用於純文字)
wc.generate(seg_str)

# 從圖片中生成顏色
image_colors = ImageColorGenerator(mask)

# 使用從圖片中生成的顏色
wc.recolor(color_func=image_colors)

# 顯示文字雲
plt.figure(figsize=(8.2, 12.49), dpi=150)
plt.imshow(wc, interpolation='bilinear')
plt.axis('off')
plt.show()
```

輸出結果：

參考資料：
[22] WordCloud for Python documentation
https://amueller.github.io/word_cloud/index.html
[23] GitHub: word_cloud
https://github.com/amueller/word_cloud

補充：OpenCC

OpenCC 是一個開源的中文簡繁轉換工具，它的特點是利用詞庫進行轉換，可以處理一些詞組翻譯上的問題，例如"鼠标"轉換為"滑鼠"而不是"鼠標"，達到與人工翻譯相媲美的效果。它支持多種轉換模式，包括簡體到繁體、繁體到簡體，以及港澳繁體到台灣繁體等等。

GitHub: https://github.com/BYVoid/OpenCC

2-4 OpenCC

```
from opencc import OpenCC

# 初始化 OpenCC，設定轉換的模式，
# 這裡設定為's2tw'（簡體到臺灣使用的繁體）
cc = OpenCC('s2tw')

# 進行轉換
to_convert = '简体中文'
converted = cc.convert(to_convert)
```

```
# 輸出轉換後的結果，應該會看到'简体中文'被轉換為'簡體中文'
print(converted)
```

Module 3. 量化文字的常見方式

One-Hot Encoding

一種表示「類別」變數的方法。在這種方法中，每一個類別都被表示為一個二進制向量，向量中只有一個元素是 1（表示該類別），其餘元素都是 0。這些向量對於不同的詞都是正交的(Orthogonal)，換句話說，One-hot 編碼無法捕捉到詞與詞之間的相似性。

	字典檔
	["人", "帥", "真好"]
人	[1, 0, 0]
帥	[0, 1, 0]
真好	[0, 0, 1]

圖：One-Hot Encoding 的範例

3-1 One-Hot Encoding

```
from sklearn.preprocessing import OneHotEncoder
import numpy as np
import jieba

# 完整的句子
sentence = "我喜歡閱讀書籍，也喜歡使用電腦來學習新的知識"

# 使用 jieba 進行斷詞
words = list(jieba.cut(sentence))

# 轉換為 OneHotEncoder 能接受的二維陣列形式
# .reshape(-1, 1) 指的是將一維陣列轉換為二維陣列，
# 其中 -1 表示自動計算列數，1 表示每列只有一個元素（代表只有一個欄位）
```

```
words_array = np.array(words).reshape(-1, 1)

# 建立 OneHotEncoder 物件
# sparse_output=False 用於指定輸出為密集陣列而非稀疏矩陣
# 例如: [[0, 1, 0], [1, 0, 0], ...]
onehot_encoder = OneHotEncoder(sparse_output=False)

# 進行 one-hot encoding
onehot_encoded = onehot_encoder.fit_transform(words_array)

# 顯示 OneHotEncoder 的分類 (類似機器學習當中的特徵名稱)
'''
範例輸出：
[array(['也', '使用', '來', '喜歡', '學習', '我', '新', '書籍', '的', '
知識', '閱讀', '電腦','，'], dtype='<U2')]

<U2 表示這些字串的最大長度為 2 個 Unicode 字元
'''
print(onehot_encoder.categories_)

# 比照原始句子，顯示 one-hot encoding 的結果
'''
原始斷詞結果
['我', '喜歡', '閱讀', '書籍', '，', '也', '喜歡', '使用', '電腦', '來',
'學習', '新', '的', '知識']
'''
print(onehot_encoded)

# 找出「我」的 One-Hot Encoding
print(onehot_encoded[0,:])

# 找出「閱讀」的 One-Hot Encoding
print(onehot_encoded[2,:])
```

詞袋模型(Bag-of-Words model, BoW)

BoW 模型將每個字詞視為詞彙表中的一個單獨條目，並將每個文件（或句

子）表示為詞彙表中各個詞頻的向量。BoW 模型忽略了詞與詞之間的順序，只考慮了詞的出現頻率。這對於一些任務可能足夠，但對於需要考慮語境的任務可能不夠，例如「我喜歡你」和「你喜歡我」在詞袋模型中是相同的。

3-2 英文部分

```
# 匯入套件
import sys
import numpy as np
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer

# 設定 numpy 顯示完整內容
np.set_printoptions(threshold=sys.maxsize)

# 詞袋模型
docs = [
    "Hey Jude, don't make it bad. Take a sad song and make it
better.",
    "Hey Jude, don't be afraid. You were made to go out and get her."
]
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(docs)
print(vectorizer.get_feature_names_out())
print(X.toarray())

# 顯示 dataframe 所有欄位
pd.set_option('display.max_columns', None)

# 透過 pandas 來預覽結果
'''
補充：
每一列代表一個文件 (document)，
它是詞頻向量 (term frequency vector)，
是一種稀疏向量表示（sparse vector representation），
稀疏、高維度、不捕捉語意，只記「這些詞出現了幾次」，
與大家常聽到大型語言模型中的 vector/embedding 概念不同。
'''
```

```
df = pd.DataFrame(X.toarray(),
columns=vectorizer.get_feature_names_out())
df
```

3-2 中文部分

```
# 匯入套件
import jieba

# 輸入文本
text = '''羅傑費德勒，已退役的瑞士男子職業網球運動員，費德勒總共贏得 20 座大滿貫冠軍，單打世界排名第一累計 310 周，其中包括連續 237 周世界排名第一的男子網壇紀錄，為網球史上最佳的男子選手之一。'''

# 分詞 (cut 是用於回傳 generator 的斷詞函式，lcut 是將斷詞以 list 格式回傳)
tokens = jieba.lcut(text)

# 去除停用詞，這邊為了簡化，我們預先定義一個停用詞列表
stop_words = ['和', '的', '是', '這']
tokens = [word for word in tokens if word not in stop_words]; tokens

# 轉換為詞向量，這裡使用詞袋模型
vectorizer = CountVectorizer()

# 轉換為詞向量矩陣
# ' '.join(tokens) 將 tokens 列表中的詞彙以空格連接成一個字串
# 因為 CountVectorizer 預期輸入是一個包含空格分隔詞彙的字串列表
li_doc = [' '.join(tokens)]
X = vectorizer.fit_transform(li_doc)

# 顯示詞彙名稱與對應的詞向量矩陣
print(vectorizer.get_feature_names_out())
print(X.toarray())

# 透過 pandas 來預覽結果
df = pd.DataFrame(X.toarray(),
columns=vectorizer.get_feature_names_out())
```

```
df
```

參考資料：

[1] NLP 的基本執行步驟(II) — Bag of Words 詞袋語言模型 https://medium.com/@derekliao_62575/nlp 的基本執行步驟-ii-bag-of-words-詞袋語言模型-3b670a0c7009

[2] Day 7 把文字裝成一袋？Bag of Word (BoW) & TF-IDF 在 NLP 中的應用 https://ithelp.ithome.com.tw/articles/10288695

TF-IDF(Term Frequency-Inverse Document Frequency)

TF-IDF 是一種統計方法，用來評估一個「詞」對於一個文件集或一個語料庫中的其中一份文件的「重要程度」。它的值越大，則詞對文件的重要性越高。

TF-IDF 是 Term Frequency（詞頻） 和 Inverse Document Frequency（逆向文件頻率）兩個詞的組合，也就是詞頻（Term Frequency，TF）和逆向文件頻率（Inverse Document Frequency，IDF）的「乘積」(TF * IDF)。

這種表示方法的特點，是可以反映出詞語對於特定文件的專屬性，比詞袋模型多了考慮字詞的稀有程度，進而能將重要的關鍵詞突顯出來。

Term Frequency (TF)	Inverse Document Frequence (IDF)
$\mathrm{tf}_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}}$	$\mathrm{idf}_i = \lg \frac{\lvert D \rvert}{\lvert \{j : t_i \in d_j\} \rvert}$

- TF：
  - 假設文件 d_j 中共有 k 個字詞，$n_{k,j}$ 是字詞 t_k 在文件 d_j 當中出現的次數，因此，分子 $n_{i,j}$ 就是該字詞在文件 d_j 中出現的次數，而分母是在文件 d_j 中所有字詞的出現次數之和。
- IDF：
  - 原則上，這裡的對數以 10 為底（以 2 為底通常是計算資訊量，以位元為單位，自然對數通常用於特定的 NLP 計算領域，例如 softmax），$|D|$ 代表語料庫中的檔案總數，$|\{j: t_i \in d_j\}|$ 包括字詞 t_j

的檔案數量（即是 $n_{i,j} \neq 0$ 的檔案數量），如果字詞不在資料當中，就導致分母為零，因此一般情況下會使用 $1 + |\{j: t_i \in d_j\}|$。

- TF-IDF 分數：
  - $tfidf_{i,j} = tf_{i,j} \times idf_i$
  - 某一特定檔案內高頻率的字詞，以及該字詞在整個檔案集合中的低頻率檔案，可以產生出高權重的 tf-idf。因此，tf-idf 傾向於過濾掉常見的字詞，保留重要的字詞。

參考網頁：

[1] 文字探勘之前處理與 TF-IDF 介紹
https://www.cc.ntu.edu.tw/chinese/epaper/home/20141220_003103.html
[2] tf-idf
https://zh.wikipedia.org/zh-tw/Tf-idf

3-3 匯入套件

```
# 匯入套件
import numpy as np
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import jieba
```

3-3 英文部分

```
# 假設我們有這些英文文本
docs = [
    "The city issued a typhoon warning and announced school closures for tomorrow; several bus routes will run on a reduced schedule.",
    "Due to the storm, multiple flights were canceled; the airport advised travelers to check updates and arrive early for security screening.",
    "Rail operators added extra trains for passenger evacuation, while highways experienced flooding and temporary road closures.",
    "The stock market turned volatile after the typhoon forecast; insurance and shipping stocks fell as investors shifted to safe-haven assets.",
```

```
    "Emergency crews inspected rivers and operated pumping stations;
residents were warned about landslides and asked to avoid mountain
areas."
]

# 建立 TF-IDF 範例
'''
ngram_range 可調整 n-gram 範圍，
(1,1) 表示只使用 unigram，
(2,2) 表示只使用 bigram,
(1,2) 表示同時使用 unigram 和 bigram
'''
vectorizer = TfidfVectorizer(
    stop_words='english', # 移除英文停用詞
    ngram_range=(1, 2), # 使用單字和雙字詞組合
)

# 進行 TF-IDF 轉換
X = vectorizer.fit_transform(docs)

# 顯示結果
print(vectorizer.get_feature_names_out())
print(X.toarray())

# 顯示 dataframe 所有欄位
pd.set_option('display.max_columns', None)

# 透過 pandas 來預覽結果
df = pd.DataFrame(X.toarray(),
columns=vectorizer.get_feature_names_out())

# 使用 TF-IDF 進行搜尋函式範例
def search_tfidf_en(query: str, top_k: int = 3, min_score: float =
0.0):
    '''
    以 TF-IDF + cosine similarity 做簡易搜尋排序。
    - query: 使用者查詢字串
```

```
    - top_k: 回傳前 k 筆
    - min_score: 低於此分數不顯示（避免全是 0 的結果）
    '''

    # 將查詢轉換為 TF-IDF 向量
    q_vec = vectorizer.transform([query])

    # 計算查詢向量與文件向量的餘弦相似度
    scores = cosine_similarity(q_vec, X).flatten()

    # 依分數由高到低排序
    ranked_idx = np.argsort(scores)[::-1]

    # 取得前 top_k 筆且分數高於 min_score 的結果
    results = []

    # 回傳 (文件索引, 分數, 內容)
    for idx in ranked_idx[:top_k]:
        if scores[idx] >= min_score:
            results.append((idx, scores[idx], docs[idx]))
    return results

# 查詢範例
queries = [
    "flight canceled airport security",
    "school closure bus schedule",
    "landslide pumping station river",
    "stock market insurance safe haven",
]
for q in queries:
    print(f"Query: {q}")
    for doc_id, score, text in search_tfidf_en(q, top_k=3, min_score=0.0):
        print(f"  score={score:.3f}  doc#{doc_id+1}: {text[:90]}...")
```

3-3 中文部分

```
# 假設我們有這些中文文本
documents = [
```

```
    "中央氣象署發布颱風警報，台北市宣布明天停班停課，部分路線公車減班。",
    "颱風影響航班取消，高鐵與台鐵加開班次疏運旅客，機場提醒提早報到。",
    "學校公告因豪雨停課，校園進行排水與樹木修剪，家長關心補課安排。",
    "股市受颱風消息影響震盪，部分保險與航運類股下跌，投資人轉向避險。",
    "市政府加強河川巡檢與抽水站運轉，呼籲民眾遠離河堤並注意土石流警戒。"
]

# 建立 TF-IDF 範例
vectorizer = TfidfVectorizer(
    tokenizer=jieba.lcut,  # 使用 jieba 進行中文分詞
    token_pattern=None,  # 使用自訂的分詞器
)

# 進行 TF-IDF 轉換
X = vectorizer.fit_transform(documents)

# 顯示結果
print(vectorizer.get_feature_names_out())
print(X.toarray())

# 透過 pandas 來預覽結果
df = pd.DataFrame(X.toarray(),
columns=vectorizer.get_feature_names_out())

# 使用 TF-IDF 進行搜尋函式範例
def search_tfidf_zh(query, top_k=3, min_score=0.0):
    q_vec = vectorizer.transform([query])
    scores = cosine_similarity(q_vec, X).flatten()
    idx = np.argsort(scores)[::-1][:top_k]
    results = []
    for i in idx:
        if scores[i] >= min_score:
            results.append((i, scores[i], documents[i]))
    return results

# 查詢範例
query = "航班取消 機場 提早報到"
```

```
results = search_tfidf_zh(query, top_k=3, min_score=0.0)
for rank, score, text in results:
    print(f"Rank {rank}: score={score:.3f}, text={text}")
```

思考：
BoW 與 TF-IDF 所建立的 vector，哪一個比較適合用來進行文件之間的相似度比對？

多數情況：

- TF-IDF 相對適合做相似度。原因是 TF-IDF 會把「到處都出現的詞」降低權重，把「更有辨識度的字詞」提升權重，讓相似度更接近你想要的「內容相似」。
- BoW 會被較高頻率的常見字詞（或長文本的總詞數）干擾，容易讓分數失真。

簡單貝葉斯分類器

簡單貝葉斯分類器（Naive Bayes）是一類以貝葉斯定理（貝氏定理）為基礎的機率式分類方法，其核心思想是：在給定類別 y 的條件下，文件中的各個特徵（例如字詞或 n-gram）彼此相互獨立。雖然「特徵獨立」在真實語言中往往不完全成立，但這個假設使得模型能以非常有效率的方式估計 $P(y \mid x)$，並利用詞彙在不同類別中的統計差異進行判斷，因此在文本分類（如情緒分析、垃圾信偵測、主題分類）中常被視為強力且可解釋的基準模型（baseline）。

在實作上，簡單貝葉斯通常搭配詞袋或 TF-IDF 等向量化方法，將文字轉為可計算的稀疏特徵矩陣，再透過平滑參數（α）避免「某詞在某類別從未出現」導致機率為零的問題，提升模型在面對未知詞彙與稀疏資料時的穩定性。

3-4 簡單貝葉斯分類器(Naive Bayes classifier)

```
import jieba
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB, ComplementNB
from sklearn.metrics import accuracy_score

# 設定隨機種子，確保實驗可重現
seed=42
```

```
# 設定資料路徑
file_path = './cases/bert_finetune/reviews.txt'

# 讀取資料
with open(file_path, "r", encoding='utf-8') as file:
    # 將每一行資料以 list 型態回傳
    lines = file.readlines()

    # 整合訓練資料
    sentences = []
    labels = []

    # 逐行讀取資料
    for line in lines:
        # 每一行資料的 tab (\t) 作為分隔符號
        parts = line.strip().split('\t')

        # 確保每一行資料都有兩個部分
        if len(parts) == 2:
            sentences.append(parts[0].strip('"'))
            labels.append(int(parts[1]))
        else:
            print(f'格式錯誤的行號: {line}')

# 切分數據集為訓練集和測試集
X_train, X_test, y_train, y_test = train_test_split(
    sentences,
    labels,
    test_size=0.2,
    random_state=seed,
    stratify=labels
)

# 建立 TF-IDF 物件
# tokenizer 預設是 None，表示使用內建的斷詞方式，這裡我們使用 jieba.lcut
來進行中文斷詞
```

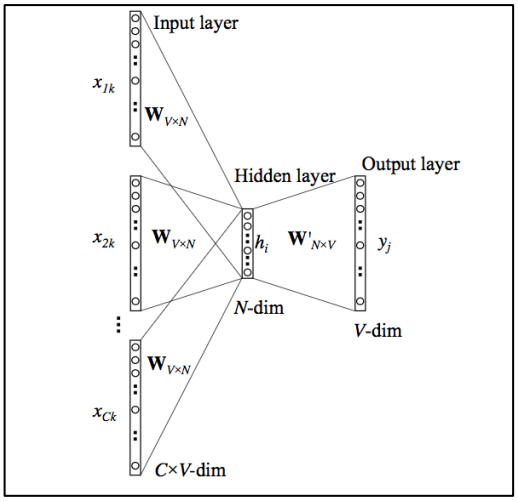
```
# token_pattern=None 是為了讓自訂的 tokenizer 生效，否則預設會以英文單字的正則表達式來切分
# ngram_range 可調整 n-gram 範圍，
# (1,1) 表示只使用 unigram，
# (2,2) 表示只使用 bigram,
# (1,2) 表示同時使用 unigram 和 bigram
vectorizer = TfidfVectorizer(
    tokenizer=jieba.lcut,
    token_pattern=None,
    ngram_range=(1, 2)
)

'''
fit(...)：從資料學出向量化規則
- 建詞表（vocabulary：有哪些詞/詞組會成為特徵）
- 計算 IDF（每個詞的 inverse document frequency）
- 決定特徵維度與欄位順序

transform(...)：用已學好的規則把新資料轉成向量
- 只把文本映射到既有詞表的欄位
- 用既有的 IDF 權重計算 TF-IDF
- 遇到訓練時沒看過的詞（OOV）就忽略（或不產生新欄位）

fit_transform(...)：
- 等於 fit() + transform() 一次做完
'''

# 對訓練集進行 TF-IDF 轉換
X_train_ft = vectorizer.fit_transform(X_train)

# 轉換測試集
X_test_t = vectorizer.transform(X_test)

# 建立貝葉斯分類器物件
# alpha: 平滑參數，預設值為 1.0，愈大表示平滑效果愈強，可避免機率為零的問題，愈小表示較少平滑，較接近原始資料分佈
'''
```

```
平滑就是把「沒看過」改成「幾乎沒看過」，例如：
- 沒平滑：沒出現 → 機率 0
- 有平滑：沒出現 → 仍給一個很小的機率
這樣模型在測試時遇到新詞或稀有詞，不會一票否決某個類別。
'''
clf = ComplementNB(alpha=1.0)

# 訓練模型
clf = clf.fit(X_train_ft, y_train)

# 進行預測
y_pred = clf.predict(X_test_t)

# 輸出準確率
accuracy_score(y_test, y_pred)
```

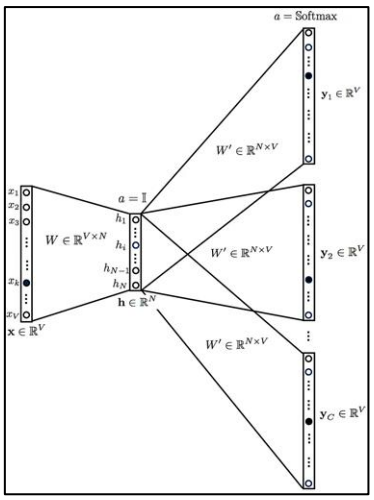
參考資料：
[1] [機器學習] MultinomialNB 貝氏(貝葉氏)分類-理論篇
https://medium.com/@d101201007/機器學習-multinomialnb-貝氏-貝葉氏-分類-df1d59b6fd9d
[2] 樸素貝葉斯分類-實戰篇-如何進行文字分類
https://www.gushiciku.cn/pl/pjta/zh-tw

詞向量(Word2Vec)

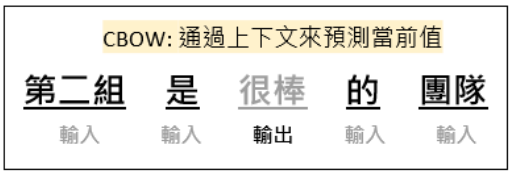
詞嵌入(Word Embedding)是一種更進階的「詞向量化」方法，它可以捕捉詞與詞之間的相關性和語義資訊。詞嵌入通常使用深度學習方法（如 Word2Vec 或 GloVe）來學習一個高維度的詞向量(Word Vector)，這個詞向量能將語義相似的字詞映射(map)到向量空間(Vector Space)的相近位置。因此，詞嵌入能比 BoW 更好地保留語言的語義結構。

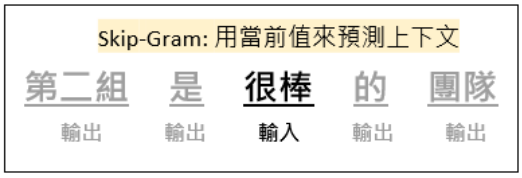


圖：Continuous Bag-of-Words，CBOW



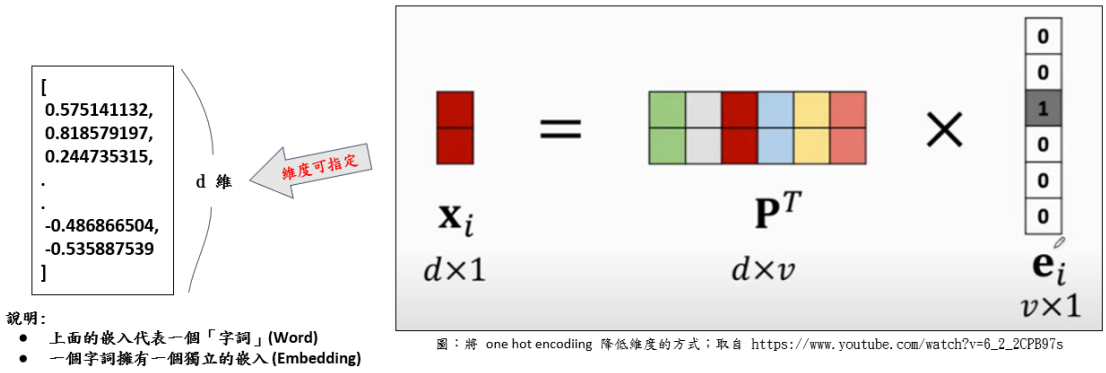
圖：Skip-Gram





CBOW 和 Skip-gram 兩者都是用「滑動視窗」建立訓練樣本，但預測方向相反：

- CBOW
  - 用上下文（context）去預測中心詞（target）
  - 特性：
    - 訓練較快
    - 對高頻率字詞表示通常更穩定
    - 大型語料時通常表現不錯
- Skip-gram（sg=1）
  - 用中心詞（target）去預測上下文（context）
  - 特性：
    - 訓練較慢
    - 對低頻率的字詞/罕見字詞通常學得比較好
- 選擇經驗
  - 語料不大（例如幾萬句以內）或你在意罕見但關鍵的詞：Skip-gram (sg=1)
  - 語料很大、你優先要速度與穩定：CBOW (sg=0)
- 沒特別設定時，Gensim 預設會用 negative sampling



圖：將 one hot encodiing 降低維度的方式；取自 https://www.youtube.com/watch?v=6_2_2CPB97s

3-5 詞向量(Word2Vec)

```
# 匯入套件
from gensim.models import Word2Vec
import jieba

# 讀取文本
documents = []
with open("./cases/bert_finetune/reviews.txt", "r", encoding="utf-8")
as file:
    for line in file:
        document = line.split("\t")[0]
        documents.append(document.replace('"', ''))

# 使用 jieba 進行斷詞 (如果是專業文本，最好自訂字典)
docs = [jieba.lcut(document) for document in documents]

# 設定參數
'''
sg = 1 -> skip-gram
sg = 0 -> cbow
'''
sg = 0

# 向前看幾個字或向後看幾個字
window_size = 5

# 向量維度
vector_size = 100
```

```python
# 訓練幾回
epochs = 5

# 最少多少個字才會被使用 (太低會帶入大量雜訊，可以嘗試 3 或 5)
min_count = 3

# 種子，用來重現訓練結果
seed = 42

# 使用幾個 CPU cores 進行訓練 (seed 只有 worker = 1 有效)
workers = 4

# 建立 Word2Vec 模型
model = Word2Vec(
    docs,
    vector_size=vector_size,
    window=window_size,
    sg=sg,
    min_count=min_count,
    workers=workers,
    seed=seed,
    epochs=epochs
)

# 取得 "房間" 這個詞的詞向量
vector = model.wv['房間']

# 輸出 "房間" 的詞向量
print(vector)

# 儲存模型
model.save('word2vec.model')

# 讀取模型
loaded_model = Word2Vec.load("word2vec.model")
```

```
# 尋找相近的字詞
loaded_model.wv.most_similar('房間', topn=10)

# 計算相近度
loaded_model.wv.similarity('房間', '房子')
```

3-5 如何找到合適的 epoch 數量?

```
from gensim.models import Word2Vec
import jieba
import matplotlib.pyplot as plt
import matplotlib
import random
matplotlib.rc('font', family='Microsoft JhengHei')

# 讀取文本
documents = []
with open("./cases/bert_finetune/reviews.txt", "r", encoding="utf-8")
as file:
    for line in file:
        document = line.split("\t")[0]
        documents.append(document.replace('"', ''))

# 使用 jieba 進行斷詞
docs = [jieba.lcut(document) for document in documents]

# 初始化模型，並啟用損失計算
model_new = Word2Vec(
    vector_size=100,
    window=5,
    sg=0,
    min_count=3,
    workers=1,
    seed=42
)

# 建立模型的詞彙表與文件標籤索引
'''
```

```
要進行訓練時，Word2Vec/Doc2Vec 需要先知道：
- 語料裡有哪些詞（words）
- 每個詞出現幾次（用來套 min_count）
- 有哪些文件標籤（tags，例如 [0], [1], [2]...）

這一步做完後，模型才知道：
- model.wv 裡有哪些詞
- model.dv 裡有哪些 doc tag（例如 5）
- model.corpus_count（語料文件數）是多少

如果你不先 build_vocab，直接 train，通常會出錯或行為不如預期。
'''
model_new.build_vocab(docs)

# 定義訓練的 epoch 數
num_epochs = 50

# 一開始的學習率
start_alpha = 0.025

# 最後的學習率
end_alpha = 0.0005

# 每跑完一個 epoch，學習率要下降多少（線性下降）
alpha_delta = (start_alpha - end_alpha) / num_epochs

# 用於儲存每個 epoch 的損失值
losses = []

# 檢視每一回合訓練後的損失值
for epoch in range(num_epochs):
    '''
    我們會用到 Stochastic Gradient Descent
    這類方法對資料順序敏感
    如果永遠同順序，可能學到偏差或收斂較差。
    打亂可讓每個 epoch 的更新路徑不同，通常更穩定。
    '''
```

```
    # 每個 epoch 開始前把文件順序打亂
    random.shuffle(docs)

    print("=" * 50)
    print(f'訓練第 {epoch+1} 個 epoch')

    '''
    a0：這個 epoch 開始時用的 alpha
    a1：這個 epoch 結束時用的 alpha

    例如：
    epoch 0：a0 ≈ 0.025，a1 ≈ 0.024...
    epoch 49：a0 ≈ 0.001...，a1 ≈ 0.0005
    '''
    # 線性更新學習率
    a0 = start_alpha - epoch * alpha_delta
    a1 = start_alpha - (epoch + 1) * alpha_delta

    # 訓練
    model_new.train(
        docs,
        total_examples=model_new.corpus_count,
        epochs=1,
        start_alpha=a0,
        end_alpha=a1,
        compute_loss=True
    )

    # 取得當前的損失值
    loss = model_new.get_latest_training_loss()

    # 得到當前的累積損失
    losses.append(loss)

    print(f"Epoch {epoch+1:02d} | alpha {a0:.5f}->{a1:.5f} | Loss
{loss}")
```

```
# 繪製損失函數圖表
plt.plot(range(1, num_epochs+1), losses)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Word2Vec 訓練損失函數圖表')
plt.show()
```

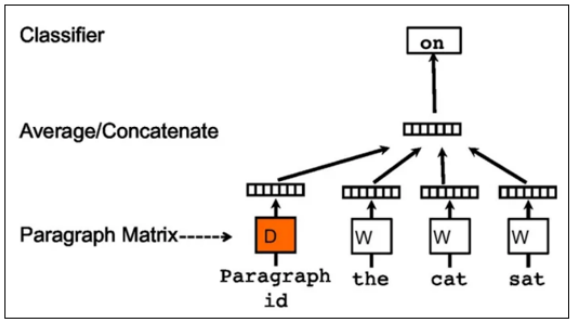
參考資料：
[1] gensim 中 word2vec 使用方法记录
https://blog.csdn.net/MarkAustralia/article/details/128466581
[2] gensim 函数库中 Word2Vec 函数 size，iter 参数错误解决
（ __init__() got an unexpected keyword argument ‘size‘）
https://blog.csdn.net/lcy6239/article/details/115786432

補充：文件向量(Doc2Vec)

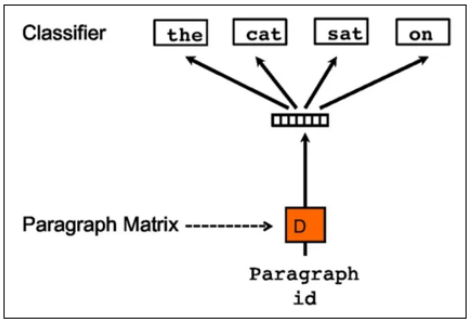
Doc2Vec 是一種將文件 (document) 轉換成數字向量的技術，這種技術能夠將文件中的字詞順序和上下文資訊編碼進向量當中。這是 Word2Vec 方法的延伸，Word2Vec 是一種將字詞轉換成數字向量的技術，它同樣能夠編碼字詞的上下文資訊。

Doc2Vec 包含兩種模型，分別是 Paragraph Vector-Distributed Memory (PV-DM) 模型和 Paragraph Vector-Distributed Bag of Words (PV-DBOW) 模型。PV-DM 模型預測目標詞語，給定上下文字詞和文件向量，而 PV-DBOW 模型則忽略上下文詞語，只用文件向量預測目標字詞。

Doc2Vec 的主要優點是它能夠學習到文件的抽象表示（一種了解上下文關係的 embedding），這些表示能夠反映文件的主題和風格。這些表示可用於各種自然語言處理任務，例如文本分類、情感分析和文件相似性比較等。



圖： PV-DM (Paragraph Vector-Distributed Memory)



圖： Paragraph Vector-Distributed Bag of Words

參考資料：https://zhuanlan.zhihu.com/p/336921474

3-6 文件向量(Doc2Vec)

```
from gensim.models import Doc2Vec
from gensim.models.doc2vec import TaggedDocument
import jieba

# 一堆句子 (也可以是一篇文章)
documents = []
with open("./cases/bert_finetune/reviews.txt", "r", encoding="utf-8")
as file:
    for line in file:
        document = line.split("\t")[0]
        documents.append(document.replace('"', ''))

# 使用 jieba 進行斷詞
docs = [jieba.lcut(doc) for doc in documents]

# 將文本標記為 TaggedDocument 格式
docs = [TaggedDocument(doc, [index]) for index, doc in
enumerate(docs)]

'''
Doc2Vec 的架構

dm=1: PV-DM (預設)
dm=0: PV-DBOW

補充：
若你用 dm=0（DBOW）但又希望同時把詞向量也學好，
通常會搭配 dbow_words=1，
因為純 DBOW 訓練主要用「文件向量去預測詞」，
詞向量可能不會被充分更新；
Gensim 的討論也指出 DBOW 的詞向量
在某些情況下可能幾乎維持初始狀態，
若需要詞向量要開 dbow_words=1。
'''
```

```
# 建立 Doc2Vec 模型
model = Doc2Vec(
    docs,
    vector_size=100,
    window=5,
    dm=1,
    # dbow_words=1
    min_count=3,
    workers=1,
    epochs=5,
    seed=42
)

# 查找索引為 5 的文件向量
vector = model.dv[5]

# 輸出索引為 5 的文件向量
print(vector)

# 顯示兩個段落的文字
print(documents[5])
print(docs[5].words)

print(documents[6])
print(docs[6].words)

# 儲存模型
model.save('doc2vec.model')

# 讀取模型
model = Doc2Vec.load('doc2vec.model')

# 查找索引為 5 的文件向量
print(documents[5])

# 找前 10 個相似的文件
similar_docs = model.dv.most_similar(5, topn=5)
```

```
print(similar_docs)
print()

for doc_index, similarity in similar_docs:
    print("=" * 50)
    print(f"Doc ID: {doc_index}, content: {documents[doc_index]}")
    print(similarity)

# 查找索引為 5 的文件向量
print(documents[0])

# 查找索引為 843 的文件向量
print(documents[843])

# 計算文檔向量 0 和文檔向量 843 之間的相似度
similarity = model.dv.similarity(5, 843)
print(similarity)
```

Module 4. 統計語言模型

簡介

統計語言模型（Statistical Language Model, SLM）是用來描述和計算一段文字在某種語言中出現機率的模型。該模型的主要目標是：給定一個詞的序列（也就是一段文字），計算這個序列出現的概率。

統計語言模型的核心是「條件機率」，也就是在一些詞已知的情況下，預測下一個詞的機率。例如，假設我們有一個詞的序列 "I am going to the"，統計語言模型的目標就是計算各種可能的下一個詞的概率，例如 "store"、"park"、"office" 等。

可以嘗試在 Google 搜尋欄位中，逐一輸入文字，它會預測使用者可以選取的文字或關鍵字。

統計語言模型有很多種類，包括以下幾種：

- **N-gram 模型**：
  這種模型假設每個字詞的出現只與前面的 N-1 個詞相關。例如，在 bi-gram 模型（N=2）中，我們假設每個字詞的出現只與前面的一個詞相關。

- **隱藏式馬可夫模型**（Hidden Markov Model, HMM）：
  這種模型假設文字是由一個隱藏的馬可夫鏈 (Markov chain) 生成的，每個字詞的出現只與前面的狀態 (state) 相關。
- **主題模型**（Topic Model）：
  這種模型假設一段文字中的每個字詞都與某種「主題」相關，並試圖找出這些主題。

N-gram 模型

簡單用法

4-1 N-gram 模型

```
import jieba

def generate_ngrams(words, n):
    # 建立空的 n-grams 列表
    ngrams = []

    # 迭代詞彙列表中的每個詞
    for i in range(len(words) - n + 1):
        # 新增下一個 n-gram
        ngrams.append(words[i:i+n])

    return ngrams

# 範例輸入
text = "我喜歡閱讀書籍，也喜歡使用電腦來學習新的知識"

# 使用 jieba 進行斷詞
words = jieba.lcut(text)

# 產生 bi-grams
bigrams = generate_ngrams(words, 2)

# 輸出結果
for bigram in bigrams:
    print(bigram)
```

計算下一個字出現的機率

4-1 N-gram 模型

```
from collections import defaultdict, Counter

# 假設我們已經有了一個已斷詞的文本列表
tokens = ['我', '愛', '吃', '蘋果', '，', '我', '也', '愛', '吃', '橘子
']

# 建立一個預設為空列表的詞典
ngram_dict = defaultdict(Counter)

# 指定我們要使用的 N-gram 的 N 值
N = 2

# 遍歷所有的詞彙
for i in range(len(tokens)-N):
    # 得到 N-gram 和它的下一個詞彙
    ngram = tuple(tokens[i:i + N])
    next_token = tokens[i + N]
    # 更新詞典
    ngram_dict[ngram][next_token] += 1

# 轉換次數為機率
for ngram, next_tokens in ngram_dict.items():
    total_count = sum(next_tokens.values())
    for next_token, count in next_tokens.items():
        ngram_dict[ngram][next_token] = count / total_count

# 假設我們要預測 "我愛" 的下一個詞彙
ngram = ('愛', '吃')

# 從詞典中取得所有可能的下一個詞彙和它們的機率
next_tokens_probs = ngram_dict[ngram]

# 將它們按照機率排序，取前 k 個
k = 2
```

```
top_k_next_tokens = sorted(next_tokens_probs.items(), key=lambda x:
x[1], reverse=True)[:k]
print(top_k_next_tokens)
```

補充：雅卡爾指數

雅卡爾指數（英語：Jaccard index），又稱交並比（Intersection over Union, IoU）或雅卡爾相似係數（Jaccard similarity coefficient），是一種用來衡量兩個樣本集合相似程度的統計量。它特別適用於有限集合的比較，其定義為兩集合「交集大小」與「聯集大小」之比：

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

4-1 Jaccard index

```
# 使用 N-gram 計算兩個句子的相似度
def jaccard_similarity(s1, s2, N):
    # 使用 jieba 進行斷詞
    words1 = jieba.lcut(s1)
    words2 = jieba.lcut(s2)

    # 產生 N-grams
    ngrams1 = generate_ngrams(words1, N)
    ngrams2 = generate_ngrams(words2, N)

    # 計算兩個句子的相似度
    common_ngrams = set(ngrams1) & set(ngrams2) # 交集
    ngrams_union = set(ngrams1) | set(ngrams2) # 聯集

    # 處理分母為 0 的情況
    if len(ngrams_union) == 0:
        return 0

    # 透過計算共同 N-grams 的數量來計算相似度
    similarity = len(common_ngrams) / len(ngrams_union)

    return similarity
```

```
# 範例
s1 = "在遙遠的東方，有一個美麗的國度，那裡山清水秀，人民勤勞智慧。"
s2 = "在很遠的東方，有一個美麗的國家，那裡山明水秀，人民勤勞聰明。"

# 使用 N-gram 來計算兩句話的相似度
similarity = jaccard_similarity(s1, s2, 2)
print(similarity)
```


參考資料：
[1] [Day4] 語言模型(一)-N-gram
https://ithelp.ithome.com.tw/articles/10259725
[2] [Day5] 語言模型(二)-N-gram 實作
https://ithelp.ithome.com.tw/articles/10259982
[3] 雅卡爾指數 Jaccard index
https://zh.wikipedia.org/zh-tw/%E9%9B%85%E5%8D%A1%E5%B0%94%E6%8C%87%E6%95%B0

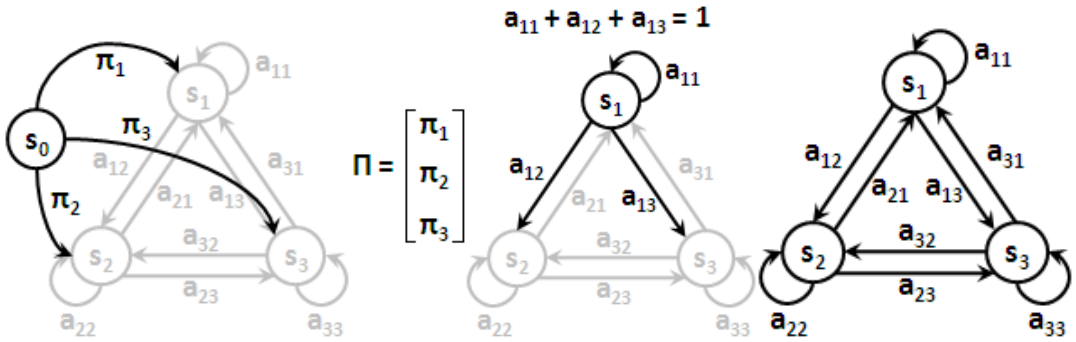
隱藏式馬可夫模型(Hidden Markov Model, HMM)

馬可夫模型描述一個系統在離散時間的狀態變化，它假設下一步只依賴目前狀態，與更久以前的歷史無關：

$$P(X_{t+1} \mid X_t, X_{t-1}, \ldots, X_0) = P(X_{t+1} \mid X_t)$$

其中 X_t 是時間 t 的狀態（state）隨機變數。

狀態集合：$S = \{s_1, s_2, \ldots, s_N\}$
狀態轉移機率（Transition Probability）：$a_{ij} = P(X_{t+1} = s_j \mid X_t = s_i)$



圖： 馬可夫模型

假設我們有 3 種狀態：$S = \{s_1, s_2, s_3\} = \{晴, 陰, 雨\}$

第一天的機率：$\Pi = \begin{bmatrix} 晴 = 0.5 \\ 陰 = 0.3 \\ 雨 = 0.2 \end{bmatrix}$，意思是第 0 天有 50% 晴，30% 陰，20% 雨。

轉移矩陣 $A = \begin{bmatrix} a11 = 0.7 & a12 = 0.2 & a13 = 0.1 \\ a21 = 0.3 & a22 = 0.4 & a23 = 0.3 \\ a31 = 0.2 & a32 = 0.3 & a33 = 0.5 \end{bmatrix}$，代表：

- 從晴天出發：到 晴/陰/雨 的機率 = 0.7/0.2/0.1
- 從陰天出發：到 晴/陰/雨 的機率 = 0.3/0.4/0.3
- 從雨天出發：到 晴/陰/雨 的機率 = 0.2/0.3/0.5

如果希望知道一開始是晴天→隔天是陰天→後天是雨天的機率，就是：

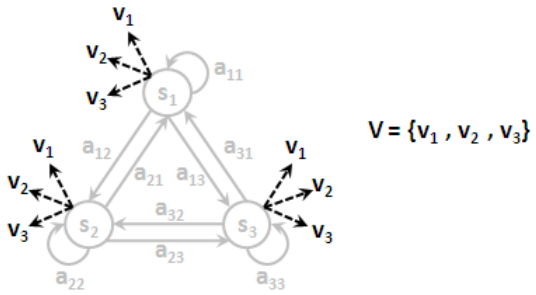
$$P(s_1, s_2, s_3) = \pi_1 \cdot a_{12} \cdot a_{23} = 0.5 \times 0.2 \times 0.3 = 0.03$$

馬可夫模型假設「狀態可觀測」，但很多現實生活中的問題：

- 真正的狀態不可直接觀測（hidden / latent）
- 我們只能看到狀態產生的觀測值（observation）

HMM 就是用來描述：

- 隱藏狀態序列 X_t 依馬可夫鏈轉移
- 每個隱藏狀態 X_t 會生成一個觀測 Y_t（如圖中的 V）



圖： 隱藏式馬可夫模型

如果我們要用隱藏馬可夫模型來計算剛才一開始晴、隔天陰、後天雨的機率，要先：

- 設定一個觀測集合 $V = \{v_1, v_2, v_3\} = \{撐傘, 戴墨鏡, 穿外套\}$。
- 定義發射機率（Emission）$b_i(v_k) = P(Y_t = v_k \mid X_t = s_i)$

再給一組直覺、每列加總為 1 的發射矩陣 B：

- 晴：戴墨鏡機率高
- 陰：穿外套機率高
- 雨：撐傘機率高

$$B = \begin{bmatrix} P(\text{撐傘}|\text{晴}) = 0.05 & P(\text{戴墨鏡}|\text{晴}) = 0.85 & P(\text{穿外套}|\text{晴}) = 0.10 \\ P(\text{撐傘}|\text{陰}) = 0.20 & P(\text{戴墨鏡}|\text{陰}) = 0.10 & P(\text{穿外套}|\text{陰}) = 0.70 \\ P(\text{撐傘}|\text{雨}) = 0.80 & P(\text{戴墨鏡}|\text{雨}) = 0.05 & P(\text{穿外套}|\text{雨}) = 0.15 \end{bmatrix}$$

一開始晴、隔天陰、後天雨的機率：

$$\pi_{\text{晴}} \cdot b_{\text{晴}}(\text{戴墨鏡}) \rightarrow \text{陰} \cdot b_{\text{陰}}(\text{穿外套}) \rightarrow \text{雨} \cdot b_{\text{雨}}(\text{撐傘})$$

$$= 0.5 \times 0.85 \times 0.2 \times 0.70 \times 0.3 \times 0.80 = 0.01428$$

4-2 隱藏式馬可夫模型

```
import sys
import numpy as np
from hmmlearn.hmm import CategoricalHMM

# 狀態與觀測名稱
STATE_NAMES = ["不喜歡", "有好感", "喜歡"]
OBS_NAMES = {1: "愉快地互動", 2: "友善回應", 3: "冷漠"}

# 建立 HMM 模型
def build_model():
    # 狀態數量與觀測值數量
    n_states, n_obs = 3, 3

    # 建立 HMM 模型物件
    '''
    n_components: 隱藏狀態數量
    n_features: 觀測值數量
    init_params: 空字串表示不進行參數初始化，使用手動設定的參數
    參考文件:
https://hmmlearn.readthedocs.io/en/latest/api.html#hmmlearn.hmm.Categ
oricalHMM
    '''
```

```
    model = CategoricalHMM(n_components=n_states, n_features=n_obs,
init_params="")

    # 設定初始機率、轉移機率、發射機率
    # 來自狀態: 不喜歡 / 有好感 / 喜歡
    model.startprob_ = np.array([0.20, 0.60, 0.20])

    # 轉移機率矩陣與發射機率矩陣
    model.transmat_ = np.array([
        [0.72, 0.23, 0.05],
        [0.12, 0.68, 0.20],
        [0.05, 0.22, 0.73],
    ])
    model.emissionprob_ = np.array([
        [0.05, 0.18, 0.77],
        [0.25, 0.60, 0.15],
        [0.72, 0.24, 0.04],
    ])

    return model

# 小劇場生成器
def story_line(state_idx: int, obs: int, step: int):
    """
    小劇場：用 (隱狀態, 觀測) 組合生成一句描述（固定模板，不依賴 random，方
便可重現）
    """
    s = STATE_NAMES[state_idx]
    o = OBS_NAMES[obs]

    # 讓敘事有點「推理感」：同一觀測在不同情緒下有不同解釋
    lines = {
        ("不喜歡", 1): [
            "她笑得很禮貌，但眼神像是在把時間切成一格一格地消耗。",
            "她的愉快像是職業反射：聲音上揚，內容卻不接球。"
        ],
        ("不喜歡", 2): [
```

```
            "她回得得體，像把話題放在托盤上遞回來，沒有多一點溫度。",
            "她很友善，但友善不等於靠近；你感覺她在維持安全距離。"
        ],
        ("不喜歡", 3): [
            "她看向窗外的時間比看向你的時間長，冷漠是一種明確訊號。",
            "她的回應短、慢、乾淨俐落，像是在結束一段不必要的對話。"
        ],
        ("有好感", 1): [
            "她主動把話題接起來，還順手丟回一顆更有趣的球。",
            "她笑出聲的瞬間很自然，像是你剛好踩到她的笑點。"
        ],
        ("有好感", 2): [
            "她的友善帶著延伸：會追問細節，也會分享自己的版本。",
            "她點頭的節奏跟你說話同步，像在說：我有在聽。"
        ],
        ("有好感", 3): [
            "她短暫冷了一下，可能是被某句話戳到雷點，也可能只是累。",
            "她沉默了一拍，但沒有逃走；更像在重新評估你這個人。"
        ],
        ("喜歡", 1): [
            "她的眼睛亮了一下，笑意不是禮貌，是失手洩漏的真心。",
            "她不只接球，還加速：把聊天變成一場雙人默契遊戲。"
        ],
        ("喜歡", 2): [
            "她很友善，而且那種友善帶著偏心：會特別記你剛剛講的小細節。",
            "她的語氣柔軟，像是已經把你放進『可以信任』的名單。"
        ],
        ("喜歡", 3): [
            "她忽然冷了一下，但不像拒絕，更像在掩飾過於明顯的在意。",
            "她把情緒收起來，可能是害羞或擔心太快被你看穿。"
        ],
    }

    # 預設句子，例如：她呈現「冷漠」，你試著推測她此刻是「喜歡」。
    # s = 狀態名稱, o = 觀測名稱
    key = (s, obs)
```

```
    # 從預設句子池中取得對應句子
    # dict.get(key, default) 的用法是: 如果 key 不在 dict 中，則回傳
default
    pool = lines.get(key, [f"她呈現「{o}」，你試著推測她此刻是「{s}」。
"])

    # 用 step 來選句子，確保同樣輸入每次結果相同
    return pool[step % len(pool)]

# 美化後驗機率輸出
def pretty_probs(p):
    '''
    後驗機率指的是在觀測到某些證據後，各隱藏狀態的機率分佈。
    輸入 p 為形狀 (3,) 的 numpy 陣列，輸出格式化字串。

    p 的內容可能類似: [0.01302611 0.49758342 0.48939047]
    代表不喜歡的機率約為 1.3%，有好感約為 49.8%，喜歡約為 48.9%。
    '''
    # 將狀態名稱與機率配對，並依機率排序
    # 例如： [('不喜歡', 0.013), ('有好感', 0.498), ('喜歡', 0.489)]
    items = list(zip(STATE_NAMES, p.tolist()))

    # 依機率由大到小排序
    # 例如： [('有好感', 0.498), ('喜歡', 0.489), ('不喜歡', 0.013)]
    items.sort(key=lambda x: x[1], reverse=True)

    # 格式化輸出
    # 例如： "喜歡 0.489 / 有好感 0.498 / 不喜歡 0.013"
    return " / ".join([f"{name} {prob:.3f}" for name, prob in items])


# 主程式
if __name__ == "__main__":
    # 建立 HMM 模型
    model = build_model()

    # 使用說明
    hint = '''\
```

```
咖啡廳 HMM 推理器
- 隱藏狀態(女生真實情緒): 0 不喜歡 / 1 有好感 / 2 喜歡
- 觀測值 (你看到的反應，請用 1/2/3 輸入):
    1 = 愉快地互動
    2 = 友善回應
    3 = 冷漠
輸入範例：1 2 2 3 2 1
離開：直接按 Enter
'''
    print(hint)

    # 主迴圈：讀取使用者輸入並進行推理
    while True:
        # 讀取觀測序列
        s = input("請輸入觀測序列(1/2/3): ").strip()

        # 離開條件
        if not s:
            print("結束。")
            break

        # 解析觀測序列
        obs_list = [int(num )for num in s.split(" ")]

        # 將觀測序列轉成 numpy 陣列 (hmmlearn 要求的格式)
        X = np.array([o - 1 for o in obs_list], dtype=int).reshape(-1,
1)

        # Viterbi：最可能的隱狀態序列
        logprob, states = model.decode(X, algorithm="viterbi")

        # Posterior：每一步各狀態的機率
        post = model.predict_proba(X)  # shape (T, 3)

        # 輸出結果摘要
        print("\n" + "-" * 72)
```

```
        print(f"你的觀測：{obs_list}  ->  {[OBS_NAMES[o] for o in
obs_list]}")
        print(f"最可能心情序列：{states.tolist()}  ->  {[STATE_NAMES[i]
for i in states]}")
        print(f"路徑對數似然 (log P): {logprob:.3f}")
        print("-" * 72)

        # 逐步輸出推理 + 小劇場
        for t, (o, st) in enumerate(zip(obs_list, states), start=1):
            print(f"[第{t:02d}步] 觀測={o}({OBS_NAMES[o]}) | 推測心情
={st}({STATE_NAMES[st]})")
            print(f"        後驗機率: {pretty_probs(post[t-1])}")
            print(f"        小劇場: {story_line(st, o, t)}")

        # 最後一步的後驗機率分析
        last_p = post[-1]

        # 找出最高機率的狀態
        best = int(np.argmax(last_p))

        # 最高機率值
        confidence = float(last_p[best])

        print("-" * 72)

        # 總結判讀 (設定門檻為 0.8 和 0.6)
        print(f"此刻(最後一步)最可能情緒：{STATE_NAMES[best]}  (後驗
={confidence:.3f})")
        if confidence >= 0.80:
            print("判讀：訊號相當明確。")
        elif confidence >= 0.60:
            print("判讀：偏向明顯，但仍可能被情境雜訊影響。")
        else:
            print("判讀：不確定性高，建議拉長觀測序列或提高觀測品質（更細的
反應類別）。")

        print("-" * 72 + "\n")
```

參考資料：

[1] Hidden Markov Model

https://web.ntnu.edu.tw/~algo/HiddenMarkovModel.html

主題模型(Topic Model)

主題模型是一種非監督學習方法，用來從大量文本中自動找出「隱含的語意主題」，並同時做到兩件事：

- 每個主題是一個字詞的機率分佈
  - 例如某個主題對「股價、央行、利率、通膨」等字詞給較高的機率，你可把它解釋成「財經」。
- 每篇文件是多個主題的混合（topic mixture）
  - 一篇文章可能 70% 談旅遊、30% 談美食；另一篇可能 50% 科技、50% 投資。

換句話說：主題模型做的是「把高維度的詞頻空間，壓縮成少數可解釋的主題維度」。

為什麼要用到 LDA（Latent Dirichlet Allocation）？

- 在文本分析裡，你常遇到這些需求：
  - 資料沒有標籤：你不知道每篇文是「科技/財經/健康」，但想先探索資料結構。
  - 想做探索與摘要：快速理解語料庫在談什麼、有哪些主要議題。
  - 想做下游任務特徵：把「每篇文的主題比例」當成特徵，用於分類、推薦、聚類、趨勢分析。
  - 想要可解釋性：比起純 embedding 的聚類，有些場景更需要「這個群為什麼聚在一起」的文字解釋（topic words）。
- LDA 是主題模型家族中最經典的版本，因為它：
  - 清楚、可解釋
  - 有完整機率模型
  - 在中大型語料上很穩定，且工具成熟

LDA 假設「文件是主題的混合、主題是詞的混合」，並從觀察到的字詞（bag-of-words）反推出每篇文的主題比例、以及每個主題的代表字詞分佈。

4-3 主題模型(Topic Model)
`from sklearn.feature_extraction.text import CountVectorizer` `from sklearn.decomposition import LatentDirichletAllocation`

```python
import jieba

# 自訂中文文件集，涵蓋多個主題
documents = [
    # ---- 科技（Tech）----
    "我把舊筆電升級到 SSD，開機速度變快，日常寫程式與資料處理更順。",
    "這款手機主打 AI 拍照與夜景模式，但續航表現與充電速度才是我在意的。",
    "智慧家居用語音控制燈光與冷氣，連網穩定度與延遲影響體驗。",
    "最近在看大型語言模型的微調方法，資料清理與評估指標很重要。",
    "顯示卡價格波動大，跑深度學習時 GPU 記憶體大小決定能否訓練。",
    "雲端服務提供彈性擴充，但成本控管與安全設定需要規劃。",
    "我在測試向量資料庫做語意搜尋，embedding 品質影響檢索效果。",
    "新出的路由器支援 Wi-Fi 6E，家裡多裝置同時連線比較不會卡。",

    # ---- 財經（Finance）----
    "台股成交量放大但波動也變高，我會分批進場並設定停損。",
    "央行升息讓房貸利率變高，現金流與風險控管變得更重要。",
    "我比較 ETF 的費用率與追蹤誤差，長期投資要看總成本。",
    "匯率變動影響出國預算，也會影響進口商品與公司毛利。",
    "財報公布後股價大震盪，市場解讀營收成長與毛利率變化。",
    "通膨數據偏高時，消費支出與物價會影響整體景氣預期。",
    "投資前我會看產業趨勢與公司競爭力，不只看短期消息。",
    "高殖利率不一定安全，還要注意現金股利是否可持續。",

    # ---- 健康（Health）----
    "最近睡眠品質差，我調整作息並減少咖啡因，精神狀態改善。",
    "健檢報告顯示血脂偏高，醫師建議飲食控制並增加有氧運動。",
    "流感季節我會戴口罩、勤洗手，也會考慮接種疫苗降低風險。",
    "久坐工作容易肩頸痠痛，我改用站立辦公並做伸展。",
    "我記錄每天走路步數與心率，觀察運動強度對體能的影響。",
    "飲食太油會影響腸胃，清淡料理搭配高纖蔬菜比較舒服。",
    "壓力大時容易頭痛，我會用冥想與呼吸練習減少緊繃。",
    "規律運動提升免疫力，但過度訓練反而可能造成疲勞與受傷。",

    # ---- 旅遊餐飲（Travel/Food）----
    "這次旅行安排了老街小吃與夜市，排隊時間會影響行程節奏。",
    "我比較住宿價格與交通便利性，靠近捷運站的飯店更省時間。",
```

```
    "去海邊玩要注意天氣與防曬，行李也要帶輕便雨具。",
    "咖啡店的手沖風味差異很大，我喜歡果香明顯的淺焙豆。",
    "烘焙麵包要控制發酵時間與溫度，口感與香氣差很多。",
    "餐廳評價不只看分數，也要看口味偏好與服務品質。",
    "安排自由行時我會先規劃景點動線，再決定交通票券方案。",
    "地方特色料理通常使用在地食材，價格合理但很看季節。",

    # ---- 混合主題文件（用來展示「一篇多 topic」）----
    "出國旅遊除了美食，匯率變動也會影響預算，我會先換好部分外幣。",
    "我想買新手機但也在考慮把預算拿去投資 ETF，兩者的成本效益要比較。",
    "跑步訓練需要運動手錶記錄心率，但產品價格與續航會影響選擇。",
    "長途旅行坐太久腰酸背痛，我會做伸展運動並挑選座位提高舒適度。",
    "雲端訂閱服務看似便宜，但月費累積後成本不低，像投資一樣要算總支出。",
    "夜市小吃很吸引人，但我會注意油脂與熱量，避免影響健康檢查數值。"
]

# 簡單停用詞（教學用：短小即可；正式可換成完整停用詞表）
stopwords = set("""
的 了 在 是 也 和 以及 與 但 更 很 我 你 他 她 它 我們 你們 他們
這 那 這次 最近 主要 內容 可能 等等 比較 影響 需要 會 先 再 我會
""".split())

# 使用 jieba 進行斷詞，將停用詞過濾掉，並將斷詞的結果以空格分隔
li_docs = []
for doc in documents:
    # 斷詞
    words = jieba.lcut(doc)

    # 過濾停用詞
    filtered_words = [word for word in words if word not in stopwords]

    # 將過濾後的詞語以空格連接成字串，加入文件列表
    li_docs.append(" ".join(filtered_words))

# 使用 CountVectorizer 進行詞頻統計，並建立詞頻矩陣
vectorizer = CountVectorizer(
    min_df=2, # 最少出現 2 次的詞才納入
```

```
    max_df=0.8, # 出現比例超過 80% 的詞不納入
    ngram_range=(1, 2) # 同時考慮一元與二元詞組
)
'''
為什麼 min_df 可以是整數也可以是浮點數？
- 當 min_df 是整數時，表示詞語在文件集中至少出現的次數。例如，min_df=2 表示詞語必須至少在 2 篇文件中出現才會被納入。
- 當 min_df 是浮點數時，表示詞語在文件集中至少出現的比例。例如，min_df=0.8 表示詞語必須至少在 80% 的文件中出現才會被納入。
同理，max_df 也是一樣的道理。
'''

# 建立詞頻矩陣
X = vectorizer.fit_transform(li_docs)

# 使用 LDA 進行主題模型訓練
lda = LatentDirichletAllocation(
    n_components=4, # 設定要找的主題數量，這裡設定為 4 個主題，你可以根據需求調整
    max_iter=100, # 最大迭代次數
    random_state=42, # 設定隨機種子以確保結果可重現
    learning_method="batch" # 使用批次學習方式，LDA 有兩種學習方式：online 和 batch，差異在於 batch 是一次使用所有文件進行更新，而 online 是分批次進行更新
)

# 訓練 LDA 模型
lda.fit(X)

# 取得詞語對應的索引
feature_names = vectorizer.get_feature_names_out()

# 顯示主題關鍵詞 (每個主題顯示前 10 個關鍵詞)
for topic_idx, topic in enumerate(lda.components_):
    # 顯示主題編號，對應 documents 中的主題
    print(f"Topic #{topic_idx + 1}:")
```

```
    # 取得該主題中權重最高的前 10 個詞語索引
    '''
    topic.argsort()[-10:][::-1]
    語法說明：
    - topic.argsort()：這會返回一個索引陣列，該陣列表示將 topic 陣列中的元素從小到大排序後的索引位置。
    - [-10:]：這會從排序後的索引陣列中取出最後 10 個索引，這些索引對應的是 topic 陣列中權重最高的 10 個元素。
    - [::-1]：這會將取出的 10 個索引反轉順序，使其從權重最高到最低排列。
    總結來說，這行語法的作用是取得 topic 陣列中權重最高的 10 個詞語的索引，並按照從高到低的順序排列。
    '''
    top_features_indices = topic.argsort()[-10:][::-1]

    # 取得對應的詞語
    top_features = [feature_names[i] for i in top_features_indices]

    # 取得對應的權重
    top_weights = topic[top_features_indices]

    # 顯示關鍵詞與權重
    for i in range(len(top_features)):
        print(f"    {top_features[i]} ({top_weights[i]:.2f})")
    print()
```

參考資料：
[1] 直觀理解 LDA (Latent Dirichlet Allocation) 與文件主題模型
https://tengyuanchang.medium.com/直觀理解-lda-latent-dirichlet-allocation-與文件主題模型-ab4f26c27184

Module 5. 深度學習與 NLP

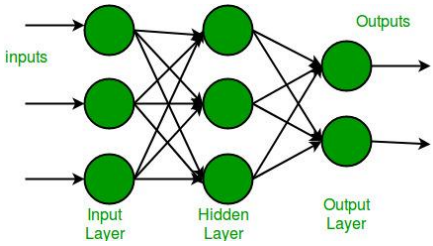
人工神經網路基礎

人工神經網路（Artificial Neural Networks，ANNs）是機器學習中的一種運算方式，這種運算方式的靈感，來自於生物的神經網路，用於模仿人腦

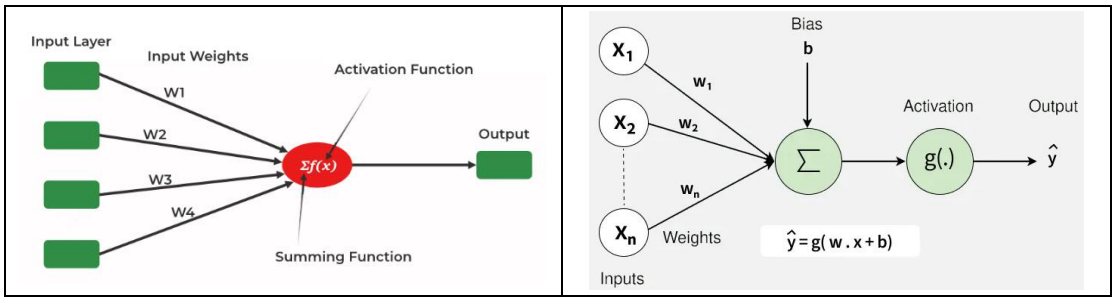
進行思考和學習的過程。

以下是對於人工神經網路在自然語言處理（NLP）中的基礎說明：

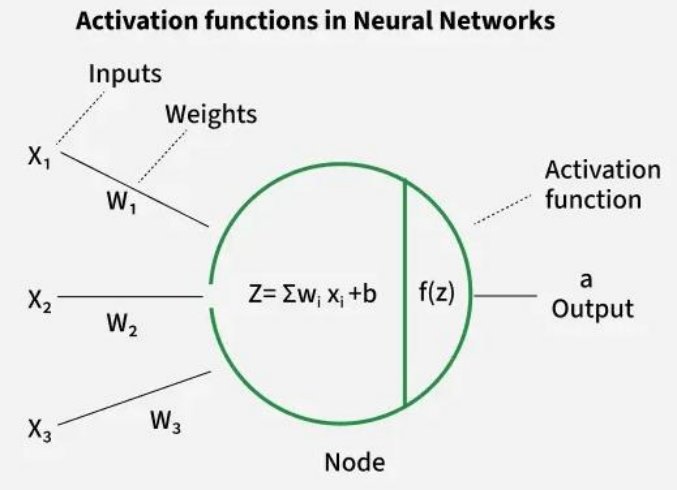
- 神經元（Neuron）：
  神經元是神經網路中的基本單元，每個神經元接收一些輸入，對輸入進行加權計算（每個輸入都有相應的權重），然後通過一個激活（也有人稱活化）函數（activation function，如 ReLU、tanh、sigmoid 等）產生輸出。
- 層（Layer）：
  神經網路由多個層組成。第一層稱為輸入層，最後一層稱為輸出層，中間的層稱為隱藏層。在 NLP 任務中，輸入通常是詞語的向量表示（如詞嵌入），輸出則取決於具體的任務（例如，在文本分類中，輸出可能是各個類別的機率）。



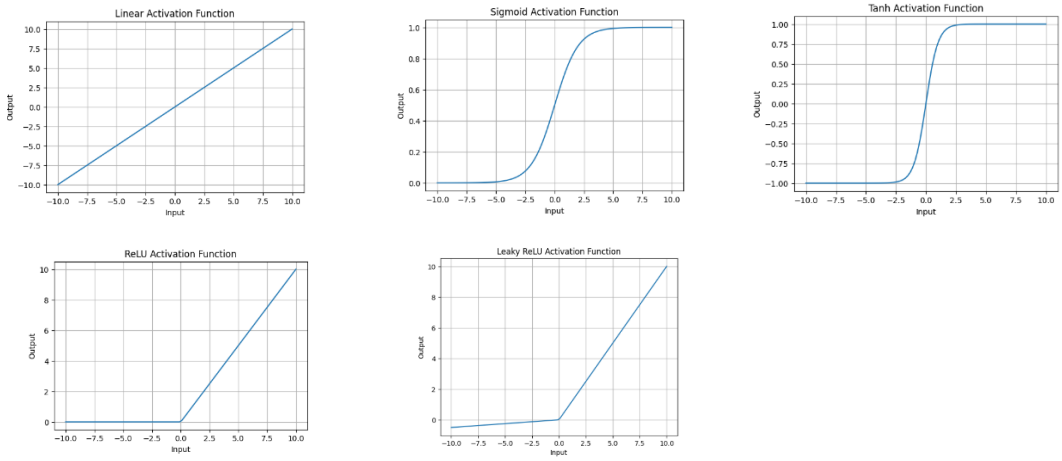
圖：機器學習中的神經網絡



圖：單層神經網路

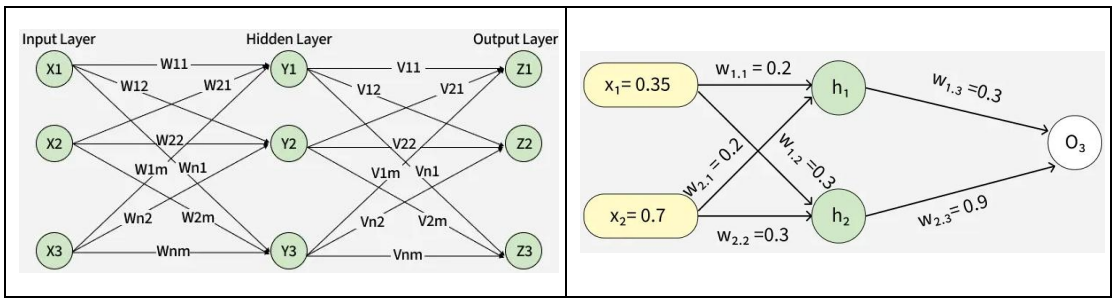


圖：神經網路中的激活函數



圖：常見激活函數

- 前向傳播（Forward Propagation）：
  神經網路的學習過程由兩個步驟組成：前向傳播和反向傳播。在前向傳播中，輸入資料通過網路向前移動，每一層都進行運算並將結果傳遞給下一層，直到輸出層。



圖：前向傳播的概念

檢視右邊例圖，試算看看（假設神經元在前向傳播和反向傳播過程中都使用 sigmoid 激活函數。目標輸出為 0.5，學習率為 1）：

1. 初始計算，每個節點的加權計算方法如下：

$$a_j = \sum (w_{i,j} * x_i)$$

其中：

a_j表示每個節點所有輸入及其權重的加權和。

$w_{i,j}$表示輸入i^{th}和神經元j^{th}之間的權重。

x_i表示i^{th}輸入的值

輸出（output），對神經元使用激活函數後，得到神經元的輸出：

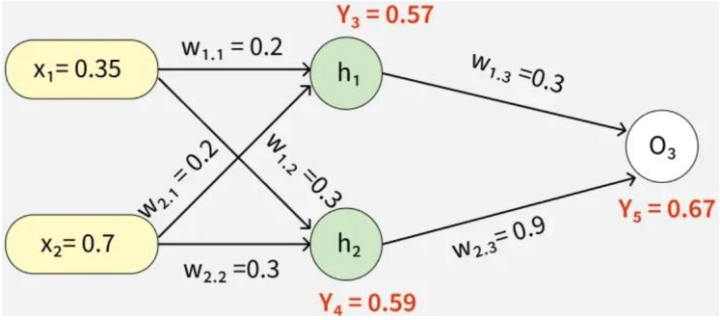
$$O_j = activation\ function(a_j)$$

2. 在這裡使用 Sigmoid Function 作為激活函數，它的計算方式如下：

$$y_i = \frac{1}{1 + e^{-a_j}}$$

它會回傳 0 ~ 1 之間的值。

3. 計算輸出



圖：計算用的參考圖片

在 h_1節點：

$$a_1 = (w_{1,1} * x_1) + (w_{2,1} * x_2)$$
$$= (0.2 * 0.35) + (0.2 * 0.7)$$
$$= 0.21$$

計算 h_1 節點 y_3 的值：

$$y_i = F(a_j) = \frac{1}{1 + e^{-a_1}}$$

$$y_3 = F(0.21) = \frac{1}{1 + e^{-0.21}}$$

$y_3 \approx 0.57$

計算 h_2 節點 y_4 的值：

$$a_2 = (w_{1,2} * x_1) + (w_{2,2} * x_2)$$
$$= (0.3 * 0.35) + (0.3 * 0.7)$$
$$= 0.315$$
$$y_4 = F(0.315) = \frac{1}{1 + e^{-0.315}}$$
$$y_4 \approx 0.59$$

計算 O_3 節點 y_5 的值：

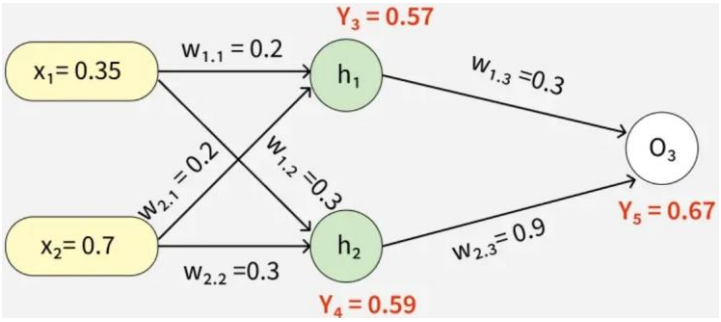
$$a_3 = (w_{1,3} * y_3) + (w_{2,3} * y_4)$$
$$= (0.3 * 0.57) + (0.9 * 0.59)$$
$$= 0.702$$
$$y_5 = F(0.702) = \frac{1}{1 + e^{-0.702}}$$
$$y_5 \approx 0.67$$

4. 誤差計算

$$Error_j = y_{target} - y_5 = 0.5 - 0.67 = -0.17$$

由於我們實際的輸出是 0.5，但計算結果是 0.67，我們將使用此誤差值進行反向傳播。

- 反向傳播（Back Propagation）和梯度下降（Gradient Descent）：
  反向傳播是一種計算梯度的方法，它是根據輸出結果和真實答案的差異（即損失函數），計算每個權重對損失的影響，然後透過梯度下降來更新權重以最小化損失。



圖：回顧先前的計算過程

1. 計算梯度

每個權重變化的計算如下：

$$\triangle w_{ij} = \eta \times \delta_j \times O_j$$

其中：

η 是學習率（learning rate）

δ_j 指的是每一個神經元（unit / neuron）在反向傳播時所對應的誤差

2. 輸出層節點的誤差

對於 O_3：

$\delta_5 = y_5 \times (1 - y_5) \times (y_{target} - y_5)$ ← 用 Chain Rule 推導出的結果

$= 0.67 \times (1 - 0.67) \times (-0.17)$

≈ -0.0376

3. 隱藏層節點誤差

對於 h_1：

$\delta_3 = y_3 \times (1 - y_3) \times (w_{1,3} \times \delta_5)$

$= 0.57 \times (1 - 0.57) \times (0.3 \times -0.0376)$

≈ -0.0027

對於 h_2：

$\delta_4 = y_4 \times (1 - y_4) \times (w_{2,3} \times \delta_5)$

$= 0.59 \times (1 - 0.59) \times (0.9 \times -0.0376)$

≈ -0.0819

4. 權重更新（帶入權重變化的計算式子）

從隱藏層到輸出層的權重：

$$\Delta w_{2,3} = 1 \times (-0.0376) \times 0.59 = -0.022184$$

新的權重：

$$w_{2,3}(new) = -0.022184 + 0.9 = 0.877816$$

從輸入層到隱藏層的權重：

$$\Delta w_{1,1} = 1 \times (-0.0027) \times 0.35 = 0.000945$$

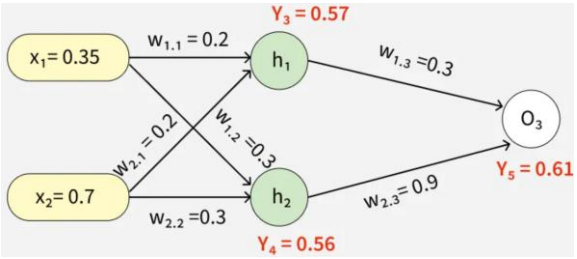
新的權重：

$$w_{1,1}(new) = 0.000945 + 0.2 = 0.200945$$

其它權重也以同樣的方式更新：

$$w_{1,2}(new) = 0.273225$$
$$w_{1,3}(new) = 0.086615$$
$$w_{2,1}(new) = 0.269445$$
$$w_{2,2}(new) = 0.185340$$

更新後的權重如下圖所示：



圖：透過反向傳播，權重得以更新

更新權重後，重複前向傳播過程，因此得到：

$$y_3 = 0.57$$
$$y_4 = 0.56$$
$$y_5 = 0.61$$

由於 $y_5 = 0.61$ 仍然不是目標輸出，因此計算誤差和反向傳播的過程將繼續進行，直到達到預期的輸出。以下過程展示了反向傳播如何透過最小化誤差來迭代更新權重，直到神經網路能夠準確地預測輸出：

$$Error_j = y_{target} - y_5 = 0.5 - 0.61 = -0.11$$

這個過程會一直持續下去，直到神經網路獲得實際的輸出結果為止。

參考資料：

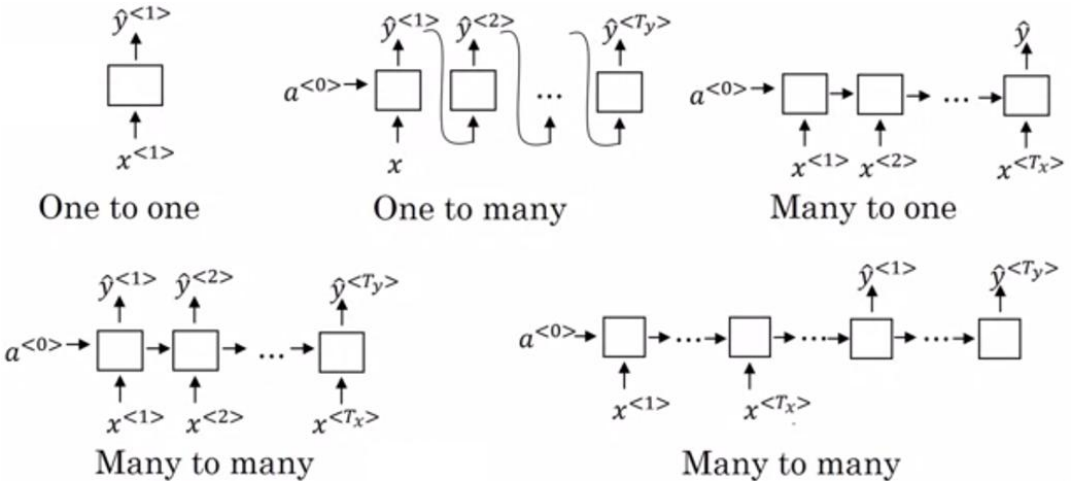
[1] 何謂 Artificial Neural Network?
https://r23456999.medium.com/%E4%BD%95%E8%AC%82-artificial-neural-netwrok-33c546c94794
[2] Deep Learning Tutorial
https://www.geeksforgeeks.org/deep-learning/deep-learning-tutorial/

序列到序列(Seq2Seq)的概念

序列到序列（Sequence-to-Sequence，簡稱 Seq2Seq）模型是一種用於

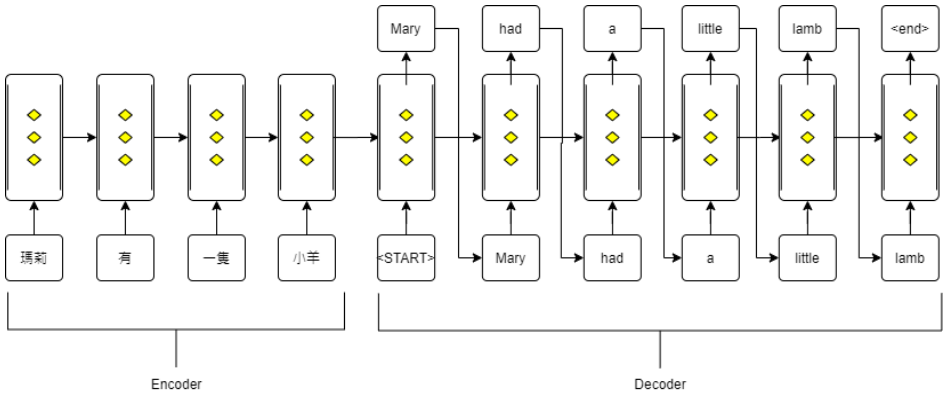
處理序列資料的深度學習架構。如其名稱所示，Seq2Seq 模型的目的是將一個序列映射到另一個序列，並且兩個序列的長度不一定相等。這使得它非常適合於如機器翻譯、語音辨識、文本摘要等任務。

架構	說明
One to one	是最基本、最傳統的神經網路類型，它對單一輸入產生單一輸出，可能是純粹的資料轉換。
One to many	是一種循環神經網路（RNN）架構，適用於單一輸入產生多個輸出的情況。音樂生成就是一個典型的應用範例。在音樂生成模型中，RNN 模型用於根據單一音符（單一輸入）產生一段音樂（多個輸出）。
Many to one	通常以情緒分析模型為例。顧名思義，這種模型用於需要多個輸入來產生單一輸出的情況。以 Twitter（現為 X）情感分析模型為例。在這個模型中，文字輸入（以單字作為多個輸入）會給予其固定的情緒傾向（單一輸出）。另一個例子是電影評分模型，該模型以評論文本作為輸入，為電影提供 1 到 5 的評分。
Many to many (1)	這指的是輸入層和輸出層大小相同的情況。也可以理解為每個輸入都有一個對應的輸出，這種情況常見於命名實體辨識（Named Entity Recognition，NER）領域。
Many to many (2)	最常見的應用是機器翻譯（Translation）。例如，英語中三個神奇的字 "I Love you" 在西班牙語中只有兩個字 "te amo"。因此，由於底層採用了大小不等的多對多 RNN 架構，機器翻譯模型能夠傳回比輸入字串多或少的字數。



圖：RNN 的不同態樣

Seq2Seq 模型主要由兩部分組成：編碼器（Encoder）和解碼器（Decoder）。每一部分都是一個循環神經網路（Recurrent Neural Network，RNN）或其他一些類型的神經網路。在機器翻譯的例子中，編碼器會接受一個語言（例如中文）的句子作為輸入，並將其轉換為一種內部表示（稱為上下文向量）。然後，解碼器將此上下文向量作為輸入，並產生另一種語言（例如英語）的句子作為輸出。

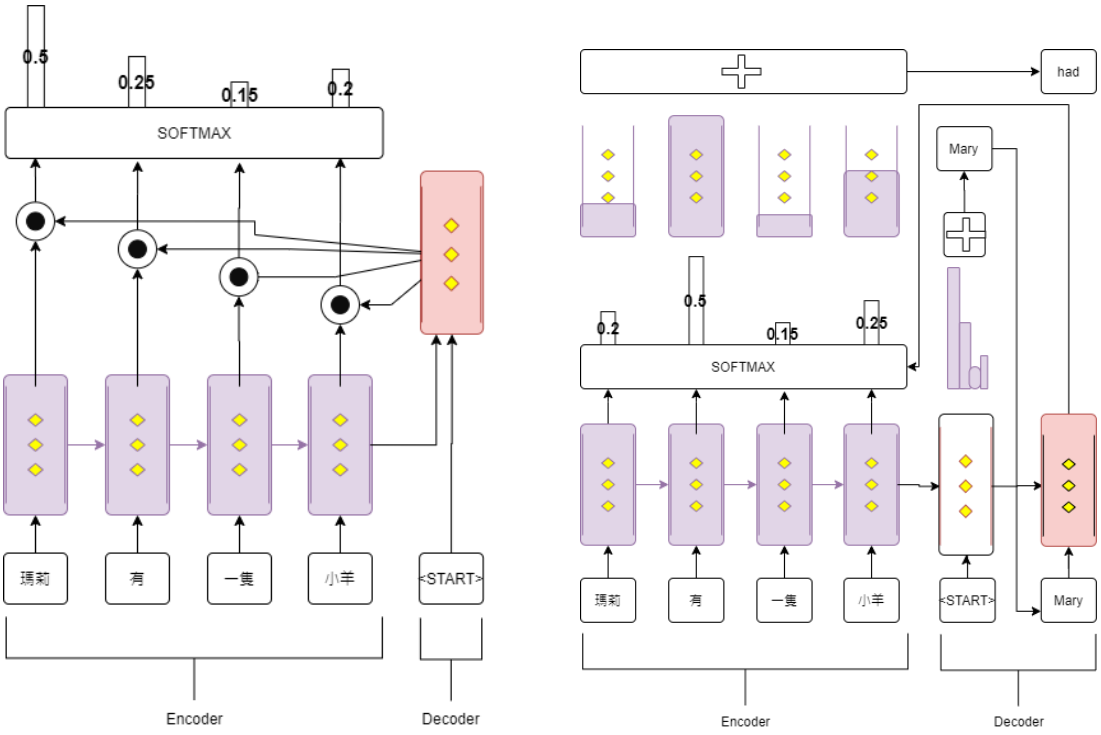


圖： Seq2Seq 架構

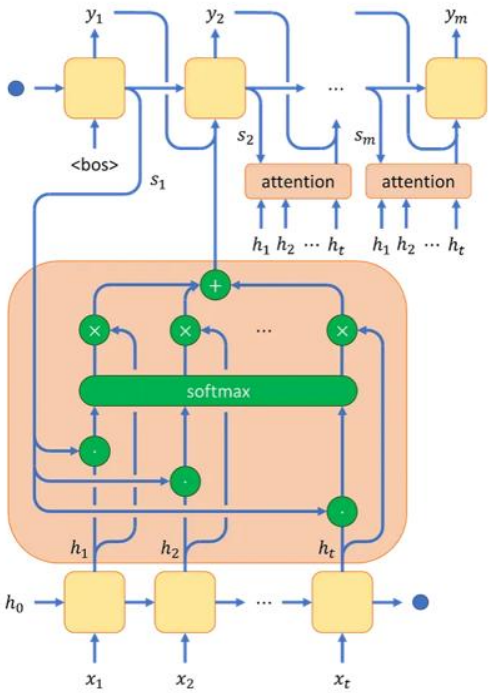
一個重要的概念是「注意力機制」(Attention Mechanism)。在基本的 Seq2Seq 模型中，編碼器需要將所有的輸入資訊壓縮到一個固定大小的上下文向量中，這可能會造成資訊的丟失。為了解決這個問題，注意力機制允許解碼器在生成每個字詞的時候，都能查看編碼器的所有隱藏狀態（而非僅看最後一個輸出），並從中選擇對當前位置最有用的狀態。這種方式顯著提高了 Seq2Seq 模型在長序列上的表現。

Seq2Seq 模型的另一個重要應用是對話系統或聊天機器人。在這種情況

下，編碼器的輸入可能是一個問題，解碼器的輸出則是對該問題的回答。



圖： 整合 attention 機制



$$H = [h_1 \quad h_2 \quad \cdots \quad h_t]^T$$

$$\text{Attention}(s_i, H) = \text{softmax}\left(\frac{s_i H^T}{\sqrt{d_h}}\right) \cdot H$$

圖： 更詳細的 attention 流程

參考資料：

[1] Types of RNN (Recurrent Neural Network)
https://iq.opengenus.org/types-of-rnn/
[2] 注意力機制（Attention mechanism）
https://medium.com/programming-with-data/28-注意力機制-attention-mechanism-f3937f289023

RNN 與 LSTM 介紹與應用

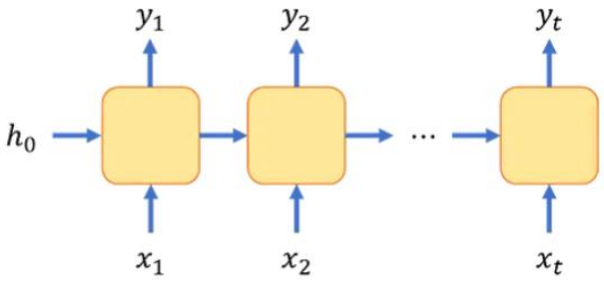
循環神經網路（Recurrent Neural Network，簡稱 RNN）

是一種特殊的人工神經網路，專門設計來處理有順序性的輸入資料。它具有記憶性，可以記住前面已處理過的資訊，並將這些資訊應用在後續的處理過程中。

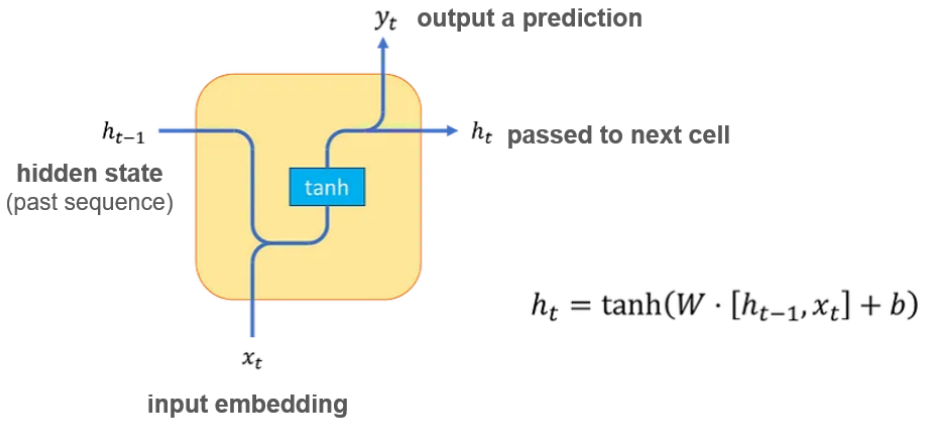
RNN 的主要結構是一個將隱藏狀態（hidden state）反饋（feed back）到自身的循環結構，這讓 RNN 能夠保存和利用過去的資訊。在每一個時間步，RNN 會接收一個新的輸入和前一時間步的隱藏狀態，然後產生一個新的隱藏狀態，這個新的隱藏狀態將會在下一時間步被用作輸入。

因此，RNN 的一個重要特性就是它具有「記憶」的能力，並且能夠使用這些記憶來處理和預測序列中的未來元素。這使得 RNN 非常適合於處理時間序列資料，如股票價格、天氣預報，或是自然語言處理中的文字和語音資料等。

然而，傳統的 RNN 存在所謂的「梯度消失（Vanishing Gradient）」和「梯度爆炸（Exploding Gradient）」問題，這使得 RNN 難以學習和理解過長的序列。為了解決這些問題，學者們提出了一些 RNN 的變體，如長短期記憶網路（Long Short-Term Memory, LSTM）和門控循環單元（Gated Recurrent Unit, GRU），它們在許多序列相關的任務中已經取得了顯著的效果。



圖： RNN 基本架構



圖：RNN 基於前一個字來理解下一個字的方式

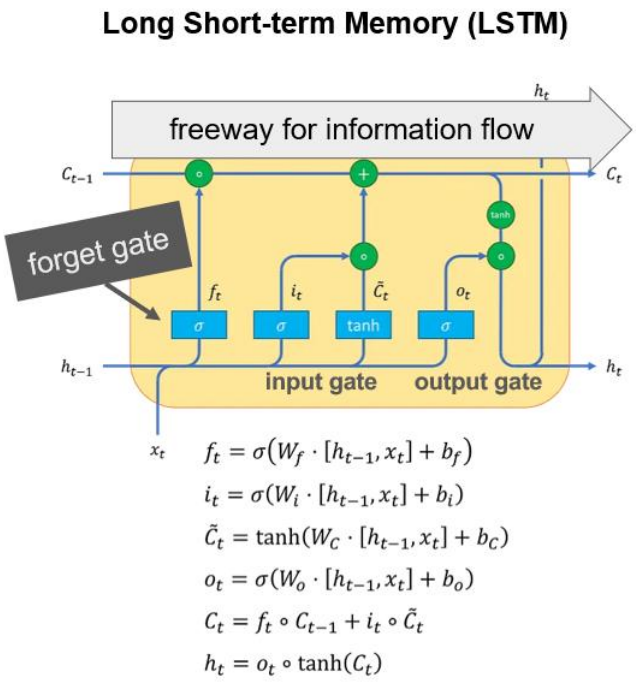
參考資料：

[1] Beautifully Illustrated: NLP Models from RNN to Transformer
https://towardsdatascience.com/beautifully-illustrated-nlp-models-from-rnn-to-transformer-80d69faf2109

LSTM（Long Short-Term Memory）

是一種特殊的循環神經網路（RNN）結構，被設計出來解決傳統 RNN 在學習長期依賴性時遇到的所謂梯度消失（Vanishing Gradient）問題。一個 LSTM 單元由三個主要部分組成，這些部分協同工作，可以讓 LSTM 單元記住或遺忘資訊：

- **遺忘門（Forget Gate）：**
  遺忘門決定 LSTM 單元應該遺忘多少過去的資訊。它使用 sigmoid 函數，可以輸出一個介於 0（忘記全部）和 1（全部保留）之間的數字。
- **輸入門（Input Gate）：**
  輸入門決定 LSTM 單元應該記住多少新的資訊。就像遺忘門一樣，它也使用 sigmoid 函數。然而，此外，LSTM 單元還將新的候選狀態（用 tanh 函數生成）與輸入門的輸出相乘，來決定哪些新的資訊應該被記住。
- **輸出門（Output Gate）：**
  輸出門決定 LSTM 單元的輸出應該是什麼。它使用 sigmoid 函數來決定哪些狀態應該被輸出，然後將這些狀態與單元狀態的 tanh 的輸出相乘，生成 LSTM 單元的最終輸出。



圖： LSTM 透過三個主要部分來決定哪些東西需要記得或忘記

這三個門的功能使 LSTM 能夠在處理長序列資料時，根據需要記住或遺忘資訊，從而有效地學習長期依賴關係。因此，LSTM 已經被廣泛地用在語音識別、語言模型、機器翻譯等自然語言處理的任務中。

LSTM 雖然能有效地解決梯度消失問題，但是仍然存在梯度爆炸問題，所以在實際操作時，還需要結合梯度裁剪（Gradient Clipping）等技巧使用。

參考資料：

[1] 簡介 LSTM 與 GRU
https://medium.com/programming-with-data/25-簡介 lstm-與 gru-3e0eaa100d29

Module 6. Transformer 神經網路模型

簡介

Transformer 是一種在自然語言處理領域中廣泛使用的深度學習模型，尤其在機器翻譯和文字生成等任務上表現優異。它的特點在於完全放棄了過去常見的循環神經網路（RNN）架構（也有人稱「遞迴式/遞歸式」神經網路），改用全新的方法處理序列資料，這使得它能夠更有效地處理長距離依賴（long-

distance dependencies）問題，並且具有很高的平行運算能力。

主要組成部分是 "自注意力機制"（Self-Attention Mechanism）或稱 "Scaled Dot-Product Attention"，它使得模型能夠權衡序列中的所有元素，並根據每個元素與其他元素的關係賦予不同的權重。這樣，模型就能夠在確定一個元素的表示時考慮到所有的上下文資訊。

另一個主要的組成部分是 "位置編碼"（Positional Encoding），因為 Transformer 本身並無法處理序列的順序資訊，因此需要通過加入位置編碼來提供該資訊。

基本結構由兩部分組成：編碼器（Encoder）和解碼器（Decoder）。編碼器由多個相同的層疊加在一起形成，每層包含一個自注意力子層（Self-Attention Neural Network）和一個全連接的前饋網路子層（Fully-connected Feedforward Neural Network）。解碼器也由多個層組成，每層還多了一個注意力子層來關注編碼器的輸出。

最著名的應用可能就是 "BERT"（Bidirectional Encoder Representations from Transformers）和 "GPT"（Generative Pretrained Transformer）等預訓練語言模型（Pre-trained Language Model），這些模型在各種自然語言處理任務上都取得了當時最好的效果。

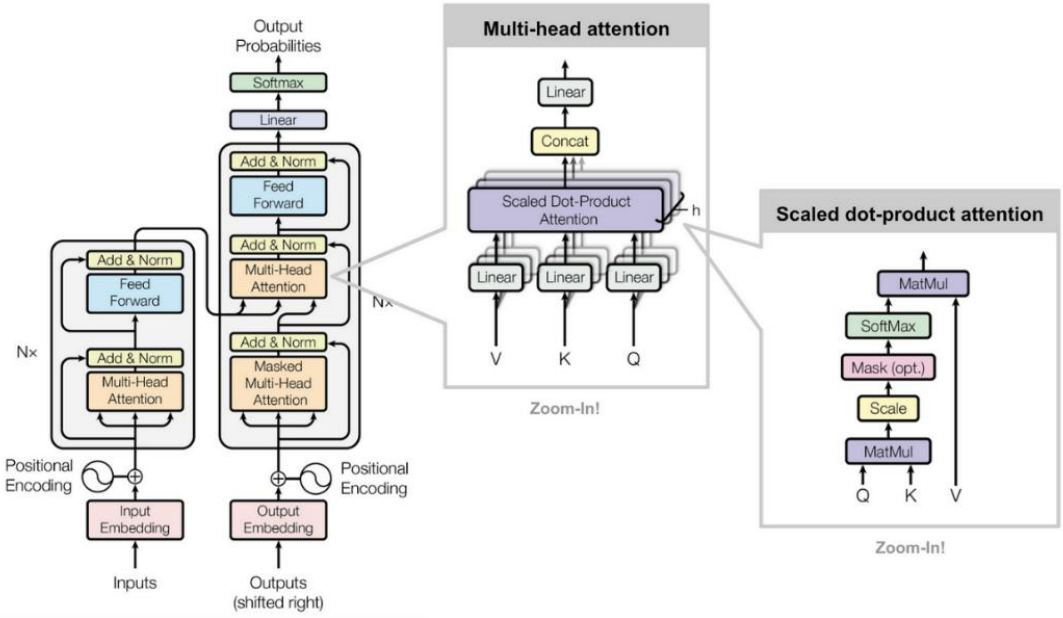
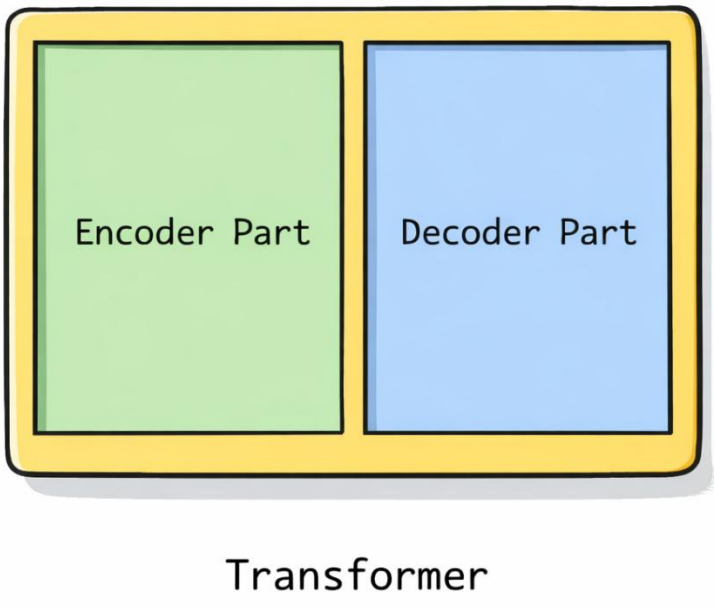


圖: Transformer 的神經網路架構（N 預設是 6）



圖：Transformers 架構分成 Encoder 與 Decoder

Q：Encoder 的輸入到底是什麼？

在 Transformer encoder 裡，每個 token 的最終輸入向量通常是：

`Input Embedding = Word (Token) Embedding + Position Embedding`

註：某些模型還會加 Segment / Token Type Embedding，但那是 BERT-like 模型的額外設計。

Q：Word Embedding 是什麼？

更精確地說是 token embedding，是把每個 token 映射成一個固定維度的向量。

- Vocab size = V
- Embedding dim = d_{model}
- Embedding matrix: $E \in \mathbb{R}^{V \times d_{model}}$

虛擬碼
"我 喜歡 NLP" ↓ tokenization [我, 喜歡, NLP] ↓ lookup [vec_1, vec_2, vec_3]

Word embedding 本身不包含順序資訊，Transformer 不能像 RNN 一樣天然知道「前後」，因此在這裡需要 Position Embedding。

假設有一句話「我 喜歡 NLP」，經過 tokenize 後，可能是

虛擬碼

```
token_ids = [101, 365, 2047] # 舉例的 token IDs
'''
Token IDs 本身沒有語意，也不是數值型特徵

它只是類似：
「請我們到 embedding table 的第 101/365/2047 列拿向量」
'''
```

接下來取得每一個 token id，到預訓練過程中所建立的向量表（Embedding table）當中，進行查詢（lookup）。

虛擬碼

```
E_word[101] → w₀ ∈ R^{d}
E_word[356] → w₁ ∈ R^{d}
E_word[2047] → w₂ ∈ R^{d}
```

組合起來就會變成：

$$X_{word} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} \in \mathbb{R}^{seq\_len \times d}$$

Q：Position Embedding（PE）是什麼？

PE =「告訴模型第幾個 token」的向量。

- 每個位置 i 和 pos 都是 zero-based
- 原始的 Transformers 使用「Sinusoidal Position Embedding」來計算 PE，它的公式如下：

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d}}\right)$$

$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

- 特性
  - 不需要學習（無參數）
  - 可外推到更長序列
  - 不同維度對不同頻率敏感
  - 相對距離可由線性組合推得

假設 pos = 2, dim = 8，那麼 position embedding 的向量就是：

虛擬碼

```
P₂ = [
```

```
  sin(2 / 10000^(0/8)), # 對應 PE 的第 0 維，也對應 Word Embedding 的第 0 維
  cos(2 / 10000^(0/8)), # 對應 PE 的第 1 維，也對應 Word Embedding 的第 1 維
  sin(2 / 10000^(2/8)), # 對應 PE 的第 2 維，也對應 Word Embedding 的第 2 維
  cos(2 / 10000^(2/8)), # 對應 PE 的第 3 維，也對應 Word Embedding 的第 3 維
  sin(2 / 10000^(4/8)), # 對應 PE 的第 4 維，也對應 Word Embedding 的第 4 維
  cos(2 / 10000^(4/8)), # 對應 PE 的第 5 維，也對應 Word Embedding 的第 5 維
  sin(2 / 10000^(6/8)), # 對應 PE 的第 6 維，也對應 Word Embedding 的第 6 維
  cos(2 / 10000^(6/8))  # 對應 PE 的第 7 維，也對應 Word Embedding 的第 7 維
]
```

經過每一個 PE 的計算，最終可以得到：

$$X_{pos} = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \end{bmatrix} \in \mathbb{R}^{seq\_len \times d}$$

Q：Word embedding 是怎麼跟 position embedding 相加的？

不是 concatenate，就是直接 add：

$$X = X_{word} + X_{pos}$$

虛擬碼

```
import numpy as np

X_word = np.array([0.1, 0.2, 0.3])
X_pos = np.array([0.4, 0.5, 0.6])
X_input = X_word + X_pos
print("X input:", X_input)
# Output: X input: [0.5 0.7 0.9]
```

從單一 token 來看：

$$x_i = w_i + p_i$$

這個 X 才是 encoder block 的真正輸入。

將 word embedding 與 position embedding 相加：

- word embedding：我是誰
- position embedding：我在第幾個位置
- 加起來：這個字（例如「NLP」）在第幾個位置（pos = 2）

Q：Encoder block 當中，Multi-Head Attention 是什麼？

我們已經知道：

$$X_{input} = X_{word} + X_{pos}$$

請參考 Transformers 的神經網路架構圖，我們先來看看 Scaled Dot-Product Attention。

在 encoder 的 attention 裡，先進行三個線性投影，可以得到 Query（Q）、Key（K）和 Value（V），Q / K / V 都來自同一個 X_{input}（所以也稱為 Self-Attention），其中：

$$Q = X_{input}W_Q，K = X_{input}W_K，V = X_{input}W_V$$

- $W_Q, W_K, W_V \in \mathbb{R}^{d_{model} \times d_k}$：可學習的線性投影矩陣（權重），都是模型參數
- $Q, K, V \in \mathbb{R}^{L \times d_k}$

想像一下：每個 token 去「問」其他 token 什麼重要，換言之，對於每個位置 i，模型會計算「第 i 個 token 應該向所有 token j 分配多少注意力權重」，然後用這些權重把各 token 的資訊加權平均，得到新的表示（representation）。

- 可以想像成
  - q_i（Query）：這個 token「想問什麼 / 想找誰」的向量
  - k_i（Key）：這個 token「我像什麼 / 我能被怎麼匹配」的向量
  - v_i（Value）：這個 token「如果你注意我，我提供什麼資訊」的向量
- 為什麼要三個？因為「怎麼比對相似度」(Q·K) 與「取用內容」(V) 可以是不同的表徵空間。

舉例而言，「大雄常常被胖虎欺負，他很害怕」這一句話，被分詞後變成「大雄 / 常常 / 被 / 胖虎 / 欺負 / ， / 他 / 很 / 害怕」等 9 個 tokens。

$$X = [x_1; x_2; \ldots; x_9] \in \mathbb{R}^{L \times d_{model}}$$

其中 $L = 9$

分詞	大雄	常常	被	胖虎	欺負	，	他	很	害怕
序號	1	2	3	4	5	6	7	8	9

把第 7 個 token「他」當成位置 $i = 7$，此時「他」這個 token 會先丟出 Query，代表「詢問『他』指的是誰？」

$$q_{他} = q_7$$

再來，「他」把自己的 Quey 跟所有 token 的 Key 計算相似度（點積，dot project），對所有 $j = 1, 2, \ldots, 9$ 進行計算：

$$s_{7j} = \frac{q_7 \cdot k_j}{\sqrt{d_k}}$$

這個「s_{7j}」的意義是：

- 「他」對每個 token j 的關聯性打分數，「大雄？胖虎？欺負？害怕？」…誰跟「他」最相關？
- $\sqrt{d_k}$ 是讓分數的尺度比較穩（一點點類似「標準化」的效果）。

之後計算 softmax，將計算結果變成注意力權重（注意力權重就是「他」在理解自己這句話時，把 "注意力" 分配到哪些詞上）：

$$a_{7j} = softmax(s_{7j})$$

且 $\sum_j a_{7j} = 1$。

虛擬碼

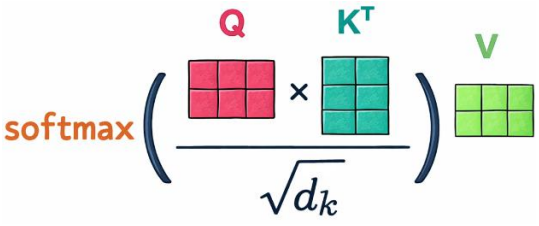
```
import numpy as np

def softmax(x):
    '''
    計算範例:

    x = [2.0, 1.0, 0.1]
    max(x) => 2.0
    x' = x - max(x) = [2.0 - 2.0, 1.0 - 2.0, 0.1 - 2.0]
    x' = [0, -1, -1.9]

    e^x' = [e^0, e^-1, e^-1.9]
    e^x' ≈ [1, 0.3679, 0.1496]
    '''
    # 計算 e 的指數次方，並減去最大值以防止數值不穩定
    e_x = np.exp(x - np.max(x, axis=-1, keepdims=True))

    # 正規化
    result = e_x / e_x.sum(axis=-1, keepdims=True)

    return result

# 範例輸入：一個 3x3 的矩陣，代表 3 個詞的注意力分數
# 這 3 個詞怎麼看呢?
```

```
# 第一列是第一個詞對其他三個詞的注意力分數，
# 第二列是第二個詞對其他三個詞的注意力分數，
# 第三列是第三個詞對其他三個詞的注意力分數
# 以下是合理的注意力分數範例
attention_scores = np.array([
    [2.0, 1.0, 0.1],
    [1.0, 3.0, 0.2],
    [0.5, 0.2, 4.0]
])

# 計算 softmax
attention_weights = softmax(attention_scores)
print("Attention Scores:\n", attention_scores)
print("Attention Weights after Softmax:\n", attention_weights)
```

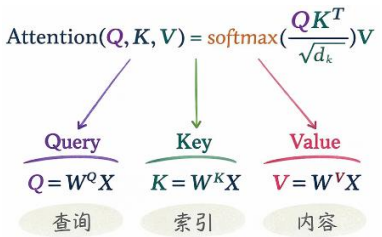
接下來用這些權重去加權平均 Value，把相關資訊「拿回來」更新「他」的表示，最後輸出「他」的新向量是：

$$o_7 = \sum_{j=1}^{9} a_{7j} v_j$$

- 如果 $a_{7,大雄}$ 的注意力分數高，那 $v_{大雄}$ 的資訊會被大量拿進來
  - 「他」的新表示會混入「大雄」的語意特徵（指代更清楚）
- 如果也關注「欺負」、「害怕」
  - 「他」的新表示也會混入「受害事件」、「情緒狀態」的資訊
  - 讓後面進行分類 / 生成的時候，更容易正確理解



圖：Self-Attention 機制



圖：實際的注意力分數計算方式

以上討論的是 Self-Attention 的概念，那 Multi-Head 呢？如果 head 的 h 值為 8，$d_{model} = 512$，那每個 head 的維度會變成：

$$d_k = d_q = d_v = \frac{d_{model}}{h} = \frac{512}{8} = 64$$

所以每個 head 看到的是 64 維的 Q / K / V。

之後每個 head 各自計算注意力：

- 第 m 個 head：
  - $Q^{(m)}, K^{(m)}, V^{(m)} \in \mathbb{R}^{L\times 64}$
- Attention scores：
  - $S^{(m)} = \frac{Q^{(m)}K^{(m)\mathrm{T}}}{\sqrt{64}}$
  - 這裡的縮放因子變成：$\sqrt{64} = 8$
- 權重：
  - $A^{(m)} = softmax(S^{(m)}) \in \mathbb{R}^{L\times L}$
- 輸出：
  - $O^{(m)} = A^{(m)}V^{(m)} \in R^{L\times 64}$

再把 12 個 head 串接回來：

$$O = Concat\left(O^{(1)}, O^{(2)}, \dots, O^{(12)}\right) \in \mathbb{R}^{L\times(8\cdot 64)} = R^{L\times 512}$$

再乘上輸出矩陣 W_o，將串接的結果投影回 d_{model}：

$$MHA(X) = OW_O, W_O \in \mathbb{R}^{512\times 512}$$

Q：之後的 Add & Norm 指的是什麼呢？

可以參考 Transformers 的神經網路架構圖，如果剛才輸出結果為：

$$Z = MHA(X) \in \mathbb{R}^{L\times d_{model}}$$

就把原先的 X 加回來（殘差）：

$$\tilde{X} = X + Dropout(Z)$$

在這裡的概念可以這麼想

- MHA 學到的是「在原本表示上該補充 / 修正哪些資訊」
- 殘差就是讓「原始訊息」保留一條捷徑，避免每層都把訊息洗掉
- 深層網路如果每層都只做變換，容易梯度消失 / 爆炸
- 殘差讓模型至少讓子層（sublayers）就算學不到有用的轉換（例如 $MHA(X) \approx 0$），也不會把原始的輸入弄壞，最保底的情況，它可以讓輸出幾乎等於輸入，這可以讓深層網路更容易訓練

接下來進 Norm，就是對每個 token 進行 LayerNorm：

$$Y = LayerNorm(\tilde{X})$$

LayerNorm（LN）是對每個 token 向量的特徵維度進行正規化。
對第 i 個 token 向量 $\tilde{x}_i \in \mathbb{R}^{d_{model}}$ 計算：

$$\mu_i = \frac{1}{d_{model}} \sum_{t=1}^{d_{model}} \tilde{x}_{i,t}$$

$$\sigma_i^2 = \frac{1}{d_{model}} \sum_{t=1}^{d_{model}} (\tilde{x}_{i,t} - \mu_i)^2$$

$$LN(\tilde{x}_i) = \gamma \odot \frac{\tilde{x}_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} + \beta$$

γ 和 β 是可學習的參數。

因此，Add & Norm 的做法就是：

$$Y = LayerNorm(\, X + Dropout(\, MHA(X)\,))$$

這個 Y 可能是下一步（進入 Feedforward Neural Networks，FFN）的輸入，或是從當前 encoder block 到下一個 encoder block 的輸入（中間表示）。

Q：進入 Feedforward Neural Networks（FFN）之後，這裡要做什麼？
FFN 主要目的就是對每個 token 的向量「各自、獨立」做一個小型兩層 MLP（非線性變換），把注意力整合後的表示，再加工成更有表達力的特徵。原始論文用的是「兩層全連接 + 非線性」，假設我們先前得到的輸出是 Y_1：

$$Z = FFN(Y_1)$$
$$FFN(x) = \max(0, xW_1 + b_1)\, W_2 + b_2$$

把它套用到每個 token 向量 $y_{1,i}$：

- 先增加維度（expand）
  - $h_i = y_{1,i}W_1 + b_1$
- 帶入非線性激活函式（這裡使用 ReLU）
  - $g_i = \sigma(h_i)$
  - 註：在高維空間做非線性，表示（representation）能力更強。
- 再降回 d_{model}
  - $z_i = g_i W_2 + b_2$

原始論文使用：

- $d_{model} = 512$
- $d_{ff} = 2048$

所以：

- $W_1 \in \mathbb{R}^{512\times2048}$
- $W_2 \in \mathbb{R}^{2048\times512}$
- $Z \in \mathbb{R}^{L\times512}$

概念上：

- Attention 子層負責把「跨 token 的資訊」整合進來，讓每個 token 知道別的 token 的關係（指代、依存、語意關聯）
- FFN 子層負責把「每個 token 自己的表示」做非線性特徵轉換，像是在每個位置做一個「特徵抽取器/分類器前的變換」，增加表示能力。

所以完整的 encoder block（N=6） 是：

- 先定義輸入
  - $X^{(0)} = X_{input} \in \mathbb{R}^{L\times d_{model}}$
- 對每一層 $\ell = 1,2,\ldots,6$ 進行
  - Multi-Head Attention 子層 + Add & Norm
    - $\tilde{X}^{(\ell)} = LN(X^{(\ell-1)} + Dropout(MHA^{(\ell)}(X^{(\ell-1)})))$
  - FFN 子層 + Add & Norm
    - $X^{(\ell)} = LN(\tilde{X}^{(\ell)} + Dropout(FFN^{(\ell)}(\tilde{X}^{(\ell)})))$
- 最後輸出 $X^{(6)} \in \mathbb{R}^{L\times d_{model}}$
  - $X^6 = \begin{bmatrix} h_1^{(6)} \\ h_2^{(6)} \\ h_3^{(6)} \\ h_4^{(6)} \\ h_5^{(6)} \\ h_6^{(6)} \end{bmatrix}$，其中每個 $h_i^{(6)} \in \mathbb{R}^{d_{model}}$

參考來源：

[1] Transformer Architecture Part -2
https://pub.towardsai.net/transformer-architecture-part-2-db1c0df20a0d

[2] transformer 中 QKV 的通俗理解(渣男與備胎的故事)
https://blog.csdn.net/Weary_PJ/article/details/123531732

[3] BertViz - Visualize Attention in Transformer Models

https://github.com/jessevig/bertviz
[4] 直觀理解 GPT-2 語言模型並生成金庸武俠小說
https://leemeng.tw/gpt2-language-model-generate-chinese-jing-yong-novels.html

BERT

BERT (Bidirectional Encoder Representations from Transformers) 是由 Google 在 2018 年提出的一種預訓練的深度學習模型，主要用於自然語言處理 (NLP) 任務。它具有很好的傳遞語境訊息的能力，並且在多項 NLP 任務中都取得了當時的最好效果。

以下是關於 BERT 的一些重要特性和概念：

- 雙向訓練：
  在傳統的語言模型中，只考慮上下文或者下文中的字詞（例如，左到右或右到左）。但是，BERT 使用的訓練方法是雙向的，它一次性考慮了上下文中的所有字詞。這對於理解語境中的每個字詞的意義有很大幫助。
- Transformer 架構：
  BERT 以 Transformer 架構為基礎，將原本用於處理序列到序列問題的 Transformer 模型調整為只包含其編碼器部分。這種架構可以處理長距離的依賴關係，並且能夠平行化處理，使得訓練更加高效。
- Masked Language Model：
  BERT 的訓練方法之一是 Masked Language Model (MLM)，其中一部分輸入的字詞會被隨機遮蔽，然後模型預測這些被遮蔽的字詞。
- Next Sentence Prediction：
  BERT 的另一種訓練方法是 Next Sentence Prediction (NSP)，模型預測第二個句子是否在原文中緊接在第一個句子之後。這種訓練方法使得 BERT 能夠理解句子之間的關係。

這些特性使得 BERT 可以用於多種 NLP 任務，例如問答系統、命名實體識別、情感分析等。尤其是在微調（Fine-tuning）後，BERT 通常能夠在這些任務中達到出色的效果。

GPT

GPT (Generative Pre-training Transformer) 是由 OpenAI 研發的語言模型，主要被應用於自然語言處理 (NLP) 的任務。

以下是 GPT 的一些關鍵特性：

- Transformers 架構：
  GPT 基於 Transformer 的架構，這種架構最初被設計為處理序列到序列的問題。但在 GPT 中，只使用了 Transformer 的解碼部分。
- 語言模型預訓練：
  GPT 是一種生成式的預訓練模型，這意味著它在大量的文本資料上進行非監督式學習，嘗試預測下一個字詞，以此學習語言的模式。然後，在特定的下游任務（如文本分類、問答等）中，通過微調(fine-tune)來適應該任務。
- 自回歸性質：
  GPT 使用了一種叫做自回歸(auto regression)的方式來生成文本。這種方式是一種自左向右的方式，也就是說，每一個字詞都只能看到它前面的字詞，不能看到後面的字詞。
- 遷移學習：
  與傳統的 NLP 模型相比，GPT 的一個主要特點是它的轉移學習能力。這種模型可以在一種任務上進行預訓練，然後在另一種不同的任務上進行微調，從而節省了大量的計算資源。
- 規模化：
  GPT 模型尤其以其規模大而著名。GPT-3，最新版的 GPT，有 1750 億個模型參數，這使得它能夠生成極為自然並符合語境的文本。

在多種 NLP 任務中，GPT 都展示了出色的性能，包括文本生成、文本分類、翻譯、摘要生成等。